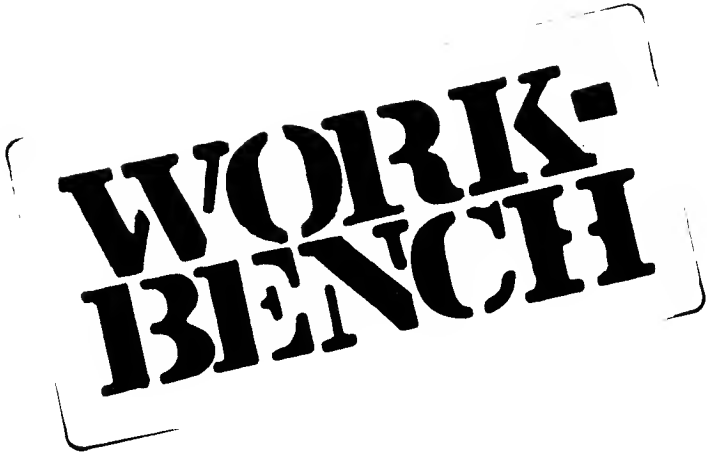
Apple

ProDOS Assembler Tools

For the Apple // Family



For use with Workbench Binder A7P0001

A2W0013

Customer Satisfaction

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Apple dealer. Return any item to be replaced with proof of purchase to Apple or an authorized Apple dealer.

Limitation on Warranties and Liability

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is," and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Copyright

Product Revisions

Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should periodically check with your authorized Apple Dealer.

© Apple Computer, Inc. 1983
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

WORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCH
WORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCH
WORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCH

ProDOS Assembler Tools

WORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCH
WORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCH
WORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCHWORKBENCH

Customer Satisfaction

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Apple dealer. Return any item to be replaced with proof of purchase to Apple or an authorized Apple dealer.

Limitation on Warranties and Liability

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is," and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Copyright

Product Revisions

Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should periodically check with your authorized Apple dealer.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Warning

This equipment has been certified to comply with the limits for a Class B computing device, pursuant to Subpart J of Part 15 of FCC Rules. Only peripherals (computer input/output devices, terminals, printers, etc.) certified to comply with the Class B limits may be attached to this computer. Operation with non-certified peripherals is likely to result in interference to radio and TV reception.

Reorder Apple Product #A2W0013

Table of Contents

List of Figures and Tables

Preface: About This Book

Preface: About This Book

Who This Book is For and What You Should Already Know

This book is for programmers who want to write assembly-language programs for the Apple II, the Apple II Plus, or the Apple IIe. All three of these computers are based on the 65Ø2 microprocessor. This book assumes that you have done some programming in BASIC or Pascal on an Apple computer system, and have read one or more books on 65Ø2 assembly-language programming (see the list of related publications at the end of this Preface).

What This Book Contains and What It Will Teach You

The ProDOS™ Assembler Tools include four programming tools that will help you create and execute assembly-language programs to run on any Apple II computer. These tools are the Editor, the Assembler, the Bugbyter debugger, and the Relocating Loader.

Each of the main chapters in this book contains an introduction to a programming tool, a brief tutorial on the use of the tool, and a detailed reference section describing how to use the full capabilities of the tool. After reading this manual and completing the short tutorials, you should be able to do the following:

- Use the **Editor** (Chapter 2) to create and modify program source files, and to store them on disk. You can also use the Editor to edit ProDOS EXEC files and BASIC program source files.

- Use the **Assembler** (Chapter 3) to generate an executable program.

- Use the **Bugbyter** debugger (Chapter 4) to test and verify the execution of your programs. You will also know how to use the Bugbyter to help you locate and fix any errors that might creep into your programs.

- Use the **Relocating Loader** (Chapter 5) to load and execute assembly language programs during the execution of a BASIC program.

The Tutorials

You will actually use the programming tools as you complete the tutorials in Chapters 2, 3, and 4. In the Editor tutorial, you will create an assembly-language source file and store it on disk. In the Assembler tutorial, you will assemble the source file and produce an executable object program. In the Bugbyter tutorial, you will test the operation of this object program and verify that it executes correctly.

Each tutorial builds on the one before, so you should go through them in sequence. Although you don't have to go through the tutorials to understand the tools described in this manual, they are the quickest way to get started programming in assembly language.

The Appendixes

Appendixes A, B, and C serve as Quick References to the Editor, Assembler, and Bugbyter.

Error messages are explained in Appendix D.

The other Appendixes contain useful information on file and table formats, using the Editor to edit ProDOS EXEC files and BASIC programs, system memory use, and Editor/Assembler file components.

Related Publications

Guides to Programming the Apple II in Assembly Language

De Jong, Marvin. Apple II Assembly Language. Indianapolis: Howard Sams, 1982. This complete manual, with an excellent introduction, was written using an early version of the Editor/Assembler. The publisher's address is 43ØØ West 62nd St., Indianapolis IN 46268.

Hyde, Randy. Using 65Ø2 Assembly Language. Northridge: DATAMOST, 1981. This thorough Apple II manual includes many tables and an introduction to the Sweet-16 ROM-coded numeric routines. The publisher's address is 19273 Kenya St., Northridge, CA 91326.

Leventhal, Lance. 65Ø2 Assembly Language Programming. Berkeley: Osborne/McGraw-Hill 1979. A guide to programming the 65Ø2, 652Ø, and 6522 microprocessors. The publisher's address is 63Ø Bancroft Way, Berkeley, CA 9471Ø.

65Ø2 Microprocessor Reference Manuals

Applications Information SY65ØØ Microprocessor Family. Santa Clara: Synertek Inc. 198Ø. A detailed pamphlet on the internal operation of the 65Ø2 microprocessor, including complete operation code timing diagrams. The publisher's address is P.O. Box 552, MS/34, Santa Clara, CA 95Ø52.

Programming Manual, MCS65ØØ Microcomputer Family. Norristown: MOS Technology, 1976. This is the standard reference for programming the 65Ø2 microprocessor, by the company that designed it. The publisher's address is 95Ø Rittenhouse Rd., Norristown, PA 194Ø1. Publication number 65ØØ-5ØA.

R65ØØ Programming Manual. Anaheim: Rockwell International Corp., 1979. An excellent alternative to the MOS Technology manual, it includes a Programming Reference Card. The publisher's address is P.O. Box 3669, Anaheim, CA 928Ø3.

65Ø2 Microprocessor Instant Reference Card. Hackensack: Micro Logic Corp., 198Ø. A comprehensive single-card chart of everything you want to know about programming the 65Ø2. The publisher's address is P.O. Box 174, Hackensack, NJ Ø76Ø2. Product number 1Ø1A.

Chapter 1

Introduction to 65Ø2 Assembly Language

Chapter 1

Introduction to 65Ø2 Assembly Language

Assembly-language programming is a powerful technique for getting the most out of your Apple computer. The ProDOS Assembler Tools will help you to create assembly-language programs that can run on any Apple II system.

The Four Programming Tools

This tool kit contains everything you need to write, assemble, and debug assembly-language programs:

- An **Editor** that lets you create and change assembly-language source programs.
- An **Assembler** that translates your assembly-language source programs into executable 65Ø2 object programs.
- The **Bugbyter** debugging program, a powerful tool for testing and debugging your programs.
- A **Relocating Loader** that allows you to load and execute your assembly-language programs during the execution of a BASIC program.

Requirements

What You Should Know

Before you continue reading this manual, you should know

- How to set up and run your Apple II system (see the owner's manual that came with the system);
- How to use ProDOS to manipulate disk files (see the manual BASIC Programming With ProDOS);
- Elementary 65Ø2 assembly-language programming concepts (see the list of Related Publications in the Preface to this manual).

What You Should Have

To use the ProDOS Assembler Tools, you need

- A computer in the Apple II family with at least 64K of RAM (random-access memory)
- A video monitor
- At least one Disk II disk drive and controller
- the ProDOS User's Manual and the ProDOS Technical Reference Manual.

A printer and a second disk drive are useful, but not required.

If You Have Other Accessories

The programming tools in this tool kit recognize which model of the Apple II family (Apple II, Apple II Plus, Apple IIe, and so on) you are using. They also recognize and make use of certain accessories that you may have installed in your system.

The system is much easier to use if you have

- An Apple IIe with an Apple IIe 8Ø-Column Text Card
- An Apple II or Apple IIe with an Advanced Logic Systems Smarterm 8Ø-Column Text Card.

The Editor and Assembler recognize the 8Ø-Column Text Card if it is installed in your Apple IIe. They also recognize the Advanced Logic Systems Smarterm 8Ø-Column Text Card if it is installed in slot 3 of your Apple II, Apple II Plus, or Apple IIe. If you have one of these cards installed in your system, the Editor/Assembler will automatically display a full 8Ø columns when you are editing or assembling your programs.

An assembly-language program designed to run on more than one Apple II configuration must be able to identify the particular member of the Apple II family on which it is running. Refer to the discussion of the System Identification Byte in the ProDOS Technical Reference Manual.

The Editor also takes advantage of the uppercase and lowercase capability of the Apple IIe computer, or of any Apple II or Apple II Plus that has the 1-wire SHIFT-key modification and can display both uppercase and lowercase characters. The Assembler can accept lowercase source files, although it does not distinguish between uppercase and lowercase except when printing.

By the Way: The Assembler Tools assume that you have a printer that prints at least 8Ø characters per line, although a printer is not required. A program with more than 2ØØ source lines will require that you print your assembly listings and use them to keep track of your programs.

Overview of Assembly-Language Programming

If you are new to assembly-language programming, the following overview briefly shows how you can use the ProDOS Assembler Tools to create working assembly-language programs for your Apple II.

An assembly-language program starts as an idea or a task that you want to accomplish. As you organize your thoughts about how your Apple can accomplish this task, you define the logic of a program. A program is merely an ordered set of detailed instructions designed to accomplish a specific task.

Translating these thoughts into a working Apple II assembly-language program involves several distinct steps:

- Using the Editor to create an assembly-language source file;
- Using the Assembler to assemble your source file, creating an executable object program;
- Using the Bugbyter program to test and debug your program;
- Finally, running your working program, either as a stand-alone program or in conjunction with a BASIC program.

Creating an Assembly-Language Source File

You will use the Editor to write your assembly-language instructions into a text file, then save this text file on disk. Chapter 2 describes how to use the Editor and includes a tutorial on editing assembly-language programs.

These text files are called assembly-language source files. A source file is not an executable program; it is just an organized file of text characters that represent assembly-language instructions and operands. In this source file, you will use a number of three-letter sequences, called **mnemonics**, to represent individual assembly-language instructions. You can explicitly represent addresses or elements of data in your program, or assign symbolic names to these addresses. These symbolic names are called **identifiers**.

Assembling Your Program

You will use the Assembler to translate your assembly-language source files into executable **object programs.** An object program is the machine-specific binary code produced by an assembler or compiler. The Assembler is actually one module of the combined Editor/Assembler program. The Assembler translates the instruction mnemonics in your source file into machine-readable codes, and translates the identifiers that you have used into the actual data or memory references that will be used by your computer. The Assembler then stores the assembled program onto disk in the form of a binary file. Chapter 3 describes the features and operation of the Assembler.

Testing and Verifying Your Program

Before trying to run an assembly-language program, you will probably want to make sure that it will execute correctly. You can use the Bugbyter program to test your program, and to fix any errors that you find. Using Bugbyter, you can easily step through any portion of your assembly-language program, checking that each instruction executes properly and that the correct data is written to the proper locations.

Bugbyter lets you see exactly what your Apple II does when it executes an assembly-language instruction. Chapter 4 describes Bugbyter in detail.

Running Assembly-Language Programs Directly From BASIC

To run an executable assembly-language program on your Apple II, simply use the ProDOS command BRUN from Applesoft BASIC. The ProDOS User's Manual describes how to use the BRUN command.

Calling an Assembly-Language Program From a BASIC Program

Assembly-language programs can be called as subroutines during the execution of a BASIC program. This type of programming combines the advantages of both BASIC and assembly-language programming. Use either the BASIC BLOAD command, or (for greater flexibility) the Relocating Loader subroutines described in Chapter 5.

WARNING
Before using the ProDOS Assembler Tools, you should make a backup copy of the system disks. The ProDOS User's Manual explains how to back up your disks.

Chapter 2

The Editor

Chapter 2

The Editor

Chapter 2

The Editor

About This Chapter

The Editor is a powerful tool for creating and modifying text files, and for storing those files on disk. Typically, you will use the Editor to create assembly-language source files that you will later use as input to the Assembler. You can also use the Editor for other purposes, such as creating and editing ProDOS EXEC files or BASIC source files.

This chapter, and the two chapters that follow it, are organized in a similar way. This chapter consists of three main parts:

- An Overview of the Editor.
- A series of brief tutorials on the use of the Editor. If you follow the directions in this chapter's tutorials, you will create an assembly-language source file that you will later use when you do the Assembler and Bugbyter tutorials in Chapters 3 and 4.
- a Reference Section that describes each of the Editor's functions in detail.

In addition, Appendix A contains two summaries of the Editor's commands: one lists them by function, the other alphabetically.

Overview

The Editor is an Apple II program that allows you to manipulate files of text that it holds in its edit buffer. This buffer is a temporary storage area that holds more than 37,ØØØ text characters, or about 1,8ØØ lines of assembly-language code.

The Editor can move and edit both individual characters and whole lines of text. The Editor treats the **line** as the basic unit of information. A line is defined as a sequence of up to 127 characters, ending with a carriage return.

Most Editor commands refer to a particular line or group of lines. The Editor keeps track of every line in your file; you refer to a particular line using that line's **relative line number.** A relative line number is a text line's position relative to the beginning of the file.

Although you will typically use the Editor to create and modify assembly-language program source files, you can also use the Editor to edit ProDOS EXEC files or BASIC program source files. See Appendix F for further information.

Tutorials

When you finish these tutorials, you will know how to

- Start the Editor
- Create a text file by typing lines of text to the Editor's text buffer
- Use the Editor commands to edit or change the text file
- Store the text file on disk
- Retrieve the text file from disk (so that you can further revise it).

By the Way: The program you will create is a very elementary one--it simply writes five characters into particular memory locations. The concepts that you use when you create, assemble, and test this very small program are worth learning, and we recommend that you do this tutorial and the ones in the next two chapters.

Getting Started

1. Insert the ProDOS Assembler Tools disk into a disk drive and turn on the power to the computer. Your Apple II loads the ProDOS operating system and briefly displays the ProDOS copyright message while loading EDASM.SYSTEM, the Editor/Assembler program's main module.
2. If you have a ProDOS-compatible clock card installed in your system, skip to Step 3. If you have none, the display shown on the next page allows you to enter the current date:

```
PRODOS EDITOR-ASSEMBLER //

ENTER THE DATE AND PRESS RETURN

DD-MMM-YY
```

- First, type the current day as two digits, using a leading zero for days less than 1Ø. The cursor skips to the month field.

- Type the three-letter abbreviation for the current month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC. The cursor skips to the year field.

- Enter the year as a two-digit number from Ø1 to 99 (zero is invalid).

- After entering the entire date, press RETURN to enter the system's Command Level.

  Because the date routine checks only that the day is less than 32 and the year is not zero, you could enter Feb 31, 19Ø1.

3. Leaving the system disk in your disk drive, press RETURN. The remainder of the Editor is then loaded from the system disk.

The Editor's Command Level

To show that you have reached the Editor Command Level, the Editor presents a ProDOS Editor/Assembler copyright message and the current date and time, followed by a colon prompt and cursor, as shown in the following display:

```
------------------------------------------------------------

   PRODOS EDITOR-ASSEMBLER //

         BY JOHN ARKLEY

        (C) COPYRIGHT 1982

        APPLE COMPUTER INC


DD-MMM-YY HH:MM
:_

------------------------------------------------------------
```

You are in the the Editor's Command Level whenever you see the colon (:) prompt character at the left margin, followed by a reverse or blinking cursor. How the cursor looks depends on which Apple II system and which, if any, 8Ø-column text card you are using.

From the Editor's Command Level, you can

- Add data to the text buffer
- Change data in the text buffer
- Write data from the text buffer to disk.

You can also invoke the Assembler from this Command Level.

Because ProDOS sets the startup prefix to /EDASM, the Assembler Tools disk must be mounted in a disk drive when the ASM command is executed.

Both the Editor and the Assembler (Assembler Tools programs EDASM.ED and EDASM.ASM) must be accessible at all times, although they are not in memory at the same time. In fact, the Editor and the Assembler are loaded alternately into the same area of memory, according to which one is needed for the operation at hand.

Using the Editor to Enter Text

When you first start the Editor, it creates a text buffer in memory.

1. To see the size of the text buffer, type

   FILE

   and press RETURN. The Editor displays three lines of data, showing the current pathname (there is none at startup), the number of bytes used in the text buffer, and the number of unused bytes remaining in the text buffer. You will see a display like the one below.

```
---------------------------------------------------------


 :FILE


     Ø BYTES USED
 37375 BYTES REMAINING
 65531 BYTES USED BY NON TEXT DATA
$2CØ4 = XSAVE AUX TYPE

:_

---------------------------------------------------------
```

2. To add some lines of text to the text buffer, type

   A

   (for Add) after the colon prompt, and press RETURN. The Editor displays 1 to show that the next line typed will be added to the Editor's text buffer as the first line of text.

3. Type the following three lines of text, exactly as they appear below. Where you see the underline (_) character, press the SPACE bar. After typing each line, press RETURN.

```
_ORG_$1ØØØ
START_LDY_#$CØ
_LDX_#Ø
```

   The Editor accepts the lines of text as you type them, and places the text into the Editor's text buffer.

4. When you have typed these lines and pressed RETURN after the last line, just press RETURN a second time without typing any other characters. Typing just RETURN (that is, a null or empty line) ends the Editor's Input mode and returns you to the Editor Command Level.

Displaying the Text

1. To view the lines of text that you have just typed into the text buffer, type

   P

   (for Print), and press RETURN. The Editor displays the text that you have typed, inserting extra spaces wherever you had typed a space character in your original text.

```
:P

                ORG   $1000
START           LDY   #$C0
                LDX   #0

:_
```

Note that the Editor interprets space characters in your text as tab characters. The Editor's default tab settings align the text in a reasonable format for assembly-language source files.

2. To display your text on the screen, together with each line's relative line number, type

   L

   (for List) and press RETURN. The Editor again displays the lines in the text buffer, this time displaying the Editor's relative line numbers along with the text, as shown on the next page.

```
:L

     1           ORG   $1ØØØ
     2 START     LDY   #$CØ
     3           LDX   #Ø

:_
```

A relative line number is the position of a line of text relative to the beginning of the text buffer; line number two is always the second line of a file, and so on. Relative line numbers have nothing to do with any characters or numbers stored within your text file. If you insert or delete lines of text within your file, the relative position of all subsequent lines changes.

By the Way: The Editor's relative line numbers should not be confused with statement numbers you might have used in BASIC programs. These BASIC statement numbers are actually stored as part of the program text. If you are using the Editor to edit a BASIC program, you will see two numbers associated with each line: the relative line number calculated by the Editor, and the statement number that is part of every BASIC statement line.

Line Editing

Perhaps you made a mistake or two when you typed your text. You can edit each line of text in turn, fixing typing errors as you go.

1. To correct errors or to add data, type

   E

   (for Edit) and press RETURN. The Editor displays the first line of your text on the screen, with the cursor over the first character:

```
1           _ORG   $1000
```

2. To edit the first line of text:

   Use the RIGHT ARROW and LEFT ARROW keys to move the cursor along the line.

   To replace a character, type the new character over the old one.

   To delete a character, place the cursor over that character and press CONTROL-D.

   To make an insertion, move the cursor to the character before which the insertion is to go, and press CONTROL-I. Type the characters you want to insert. When you finish inserting characters, press RIGHT ARROW or LEFT ARROW.

3. Edit your text until it is exactly what you were instructed to type. If there are no errors in your text (good for you!) press RETURN until you are back in the Editor Command Level.

   As you press RETURN after each line, the Editor incorporates all the changes you have made to the displayed line, and then displays the next line for you to edit.

4. When you are satisfied with the line as it appears on the screen, press RETURN.

Storing and Retrieving Files

Having returned to the Editor Command Level, you can save your text on the system disk.

1. To save your text, type

   ```
   SAVE TESTPROGRAM
   ```

   and press RETURN. The Editor saves your text in a new file named TESTPROGRAM.

2. To verify that your text file was created correctly, type

```
CAT
```

(for CATalog) and press RETURN. The Editor displays a list of the files on the disk, including the file TESTPROGRAM, as shown below:

```
:CAT

/EDASM
 NAME          TYPE  BLOCKS  MODIFIED

 PRODOS         SYS      28  2Ø-JAN-83
 EDASM.SYSTEM   SYS       8  2Ø-JAN-83
 EDASM.ED       BIN      16  2Ø-JAN-83
 EDASM.ASM      BIN      28  2Ø-JAN-83
 BASIC.SYSTEM   SYS      15  2Ø-JAN-83
 TESTPROGRAM    TXT      Ø1  dd-mmm-yy

 :_
```

Each file in the catalog has a TYPE. Note that TESTPROGRAM's type is TXT (for TeXT). The Editor's SAVE command creates only text files; your assembly-language program is a standard ProDOS sequential text file. Each file is also dated; TESTPROGRAM has been given the date you entered when you loaded the Editor.

3. To confirm that your text is still in the Editor's text buffer, type

```
L
```

(for List) and press RETURN. The Editor displays the contents of its text buffer, showing that your file is still there (as well as on the disk).

4. To clear the text buffer, type

```
NEW
```

and press RETURN. The Editor clears the text buffer, so that you can type new text or perform some other function. If you

then type

L

and press RETURN, the Editor will show you that there is no text in the buffer.

5. To load a text file that has been saved on the disk, type

LOAD

followed by a space and the text file's name, and press RETURN. (For example, to reload the text file that you just saved, type

LOAD TESTPROGRAM

and press RETURN.)

6. To see that your text is now back in the Editor's text buffer, type

L

and press RETURN. The Editor displays the text on the screen, exactly as it looked before you saved it.

Writing a Program With the Editor

You have written the first three lines of an assembly-language source file and saved it in the Editor's text buffer. To finish this program, you will add more lines to the text buffer and then save the complete program on disk.

1. To begin appending lines to the program, type

A

and press RETURN. Note that the relative line number 4 appears before the prompt. You are now ready to enter the rest of your program, starting with line 4.

2. Type the lines shown on the following page. Before pressing RETURN, make sure that the line is typed exactly as it appears in the illustration, including spaces. To leave input mode, press RETURN twice after the last line.

```
LOOP_JSR_STORE
_INX
_CPX_#5
_BNE_LOOP
_RTS
STORE_INY
_TYA
_STA_BUFF,X
_RTS
_DS_$ED,$ØØ
BUFF_DS_1Ø,$ØØ
```

3. To List what you've typed, type

```
L
```

and press RETURN. The Editor displays the text lines that you have typed.

Your program should now appear exactly as shown below.

```
1             ORG   $1ØØØ
2 START       LDY   #$CØ
3             LDX   #Ø
4 LOOP        JSR   STORE
5             INX
6             CPX   #5
7             BNE   LOOP
8             RTS
9  STORE      INY
1Ø            TYA
11            STA   BUFF,X
12            RTS
13            DS    $ED,$ØØ
14  BUFF      DS    1Ø,$ØØ
```

4. If there are any mistakes, use the Line Editing commands to correct them.

5. When you have finished, type

        SAVE

   and press RETURN. The Editor saves your completed assembly-language program on the disk, automatically using the name TESTPROGRAM.

Leaving the Editor

To leave the Editor and return to BASIC, type

        EXIT

and press RETURN. You should now see the BASIC prompt (]).

You have now finished the first tutorial, in which you have learned how to use the Editor to do a variety of tasks. You can now

- Start up the Editor
- Enter text into the Editor's text buffer
- Edit the text in the text buffer
- Display the text in the text buffer
- Save a text file on disk
- Retrieve the text file from disk, in order to edit it and save it again.

You will use your new TESTPROGRAM when you do the tutorials in the Assembler and the Bugbyter chapters.

Reference Section

The preceding tutorial introduced you to some of the basic features of the Editor. The remainder of this chapter discusses them in detail, and describes other features not mentioned in the tutorial.

The Editor Command Level

You are in the the Editor's Command Level whenever the colon (:) prompt is displayed at the left margin, followed by the cursor. You can access all of the Editor features by typing commands from this Command Level.

By the Way: The cursor can be an underscore, a box, or some other character, depending on the specific Apple II system and/or 8Ø-column text card installed.

Getting Help

When first learning to use the Editor, you may find it difficult to remember all the commands and syntax. Fortunately, the Editor provides a built-in "reference card" to remind you of these commands and their syntax.

To view this "reference card" when you are using the Editor, type

```
?
```

after the colon prompt and press RETURN. The Editor then shows you the first part of a three-part list of the Editor commands, together with the most common syntax for each command. To see the second and third parts, press any character key twice. If you are fast, you can single-step the reference card display with the SPACE bar or cancel it with CONTROL-C. You can print the help display; review the PTRON and PTROFF commands in the section "Using a Printer With the Editor."

Abbreviating Editor Commands

When you type commands to the Editor, you can often abbreviate the command names to just one or two letters. The Editor ignores any spaces that you insert before the command or between the command and its parameters.

Typing More Than One Command Per Line

If you type more than one Editor command on a single line, you must separate the commands with colons (:).

Use caution when typing more than one command on a command line. The Editor executes the commands in sequence. If an error occurs, all unused commands are discarded. In addition, if the command lines were being executed from an EXEC file, the EXEC file will be terminated as well. (A failure to locate the target character or string in a Change, Find, or Edit search is not considered an error.)

Relative Line Numbers: A Warning

Whenever you type the Add, Delete, Insert, Copy, or Replace commands, the relative line numbers of all subsequent lines in your text may be changed. The Editor uses the new relative line numbers when executing any subsequent commands. Always check the relative line numbers of any text before you Replace or Delete it. Checking is especially important if you type more than one command per command line.

Time and Date

Whenever you enter a null line at the Editor Command Level, the Editor calls the ProDOS clock routine and displays the date. If your system has a clock card that ProDOS can use, the current time is also displayed.

Typing Uppercase and Lowercase Characters

You can insert lowercase characters in your text (for example, to have your program display a message in lowercase). The method depends on your particular Apple II system.

No special commands are needed if you have

- An Apple IIe computer or a more recent member of the Apple II family
- An Apple II or Apple II Plus computer with the one-wire shift key modification installed, and an ALS Smarterm 8Ø-Column Text Card in Slot 3.

If you have an Apple II or Apple II Plus computer with the one-wire shift key modification installed, but not the ALS Smarterm 8Ø-Column Text Card:

- You must indicate to the Editor that your SHIFT key is active by typing

  SET Lcase

  and pressing RETURN.

- After enabling entry of lowercase characters, you can disable it by typing

  SET Ucase

  and pressing RETURN.

If you have an Apple II or Apple II Plus computer that does not have a keyboard shift-key modification to allow you to type lowercase characters, you can use two of the Editor's commands to act as software shift keys. When you first start the Editor, it is set to accept uppercase letters only.

- To enable the entry of lowercase characters, press CONTROL-E and RETURN.

- To disable lowercase, press CONTROL-W and RETURN.

By the Way: Although the Editor recognizes commands typed in either uppercase or lowercase, character strings are all sensitive to case. ProDOS recognizes "EditA" and "edita" as the same pathname. On the other hand, these same two strings are considered different strings when the Editor is searching the text buffer for a Find, Change, or Edit command.

Accessing Disk Volumes and Directories

The ProDOS Prefix

The Editor defines, changes, and clears the ProDOS prefix. The ProDOS prefix determines which disk (or other mass storage volume) is searched for the current directory or subdirectory. Each ProDOS disk has a "volume name" or "root directory name." A root directory may contain multiple levels of subdirectories.

The remainder of this section assumes that you understand certain fundamental ProDOS concepts. If you are not familiar with the ProDOS prefix, or with volumes and directories, please review the relevant sections in your ProDOS User's Manual.

Displaying the Online Volume Names

To load files from disk, you need to know the complete pathnames to those files. The volume name is an essential part of the pathname. To see a list of the volume names of the disks that are in your disk drives, type

```
ONLINE
```

and press RETURN.

ProDOS searches each available disk drive and reads the root directory name or volume name from the disk. The Editor then displays the list of online volume names.

The Current Prefix

When you load a file from disk, or save a file to disk, ProDOS prefixes the pathname you give with the current ProDOS prefix (unless you start your pathname with a slash, indicating that you are supplying a complete pathname that includes a volume name, optional directory names, and a filename).

To display the current ProDOS prefix, type

```
PreFiX
```

and press RETURN. This form of the PreFiX command causes the current ProDOS prefix to be displayed.

The three files EDASM, EDASM.ED, and EDASM.ASM must all reside on the same disk volume and within the same directory. This disk volume must be in a drive when the Assembler is invoked with the ASM command. The system will prompt you to insert this disk if it cannot be found when an assembly is completed. EDASM may be invoked using the dash command from the ProDOS BASIC interpreter.

Changing the Current Prefix

To change the current ProDOS prefix, type

```
PreFiX
```

and the appropriate pathname, and press RETURN. The pathname must not

contain embedded spaces or any characters that are invalid for a pathname. (Use the Online command to find out what volume names are available.)

Error Messages: The error message BAD PATH/FILE NAME means that the pathname you provided has invalid syntax. If the pathname syntax is valid, but the pathname is nonexistent, the message DIRECTORY NOT FOUND or PATH NOT FOUND appears, depending on what was misspelled.

To restore the current ProDOS prefix to the startup prefix, type

```
PFX/
```

and press RETURN. This command syntax requests that the internal startup prefix, retained by EDASM at boot time, become the current prefix. The current user prefix becomes the startup prefix and it is displayed as though the normal PREFIX pathname syntax had been used.

The Editor remembers the pathname that you typed when you last loaded your file or when you last saved your file. This pathname is called the **current pathname**, and is used by the Editor to save you from having to retype the pathname every time you want to back up your file onto disk.

To view the Editor's current pathname, type

```
FILE
```

and press RETURN. The Editor displays the current pathname, along with the length of the text in the buffer and the amount of space remaining in the buffer, both in bytes. The sum of these two numbers is the total memory currently available to the Editor. (If you have not yet executed a Load or Save command during this edit session, there is no current pathname associated with your text.)

Viewing Disk Directories

Whenever you are using the Editor, you can view the current (as determined by the current prefix) disk directory.

To see the short form (40 columns) of the current directory, type

```
CAT
```

and press RETURN.

To see the wide form (8Ø columns) of the current directory, type

CATALOG

and press RETURN.

To see a directory or subdirectory other than the current one, type

CAT

and the pathname to any available root directory or subdirectory, and press RETURN. (This will not change the current prefix.)

Saving and Retrieving Text Files

The Editor is not limited to working only with assembly-language source files -- in fact, it can work with any ProDOS sequential text file that will fit in the Editor's buffer space. You can use the Editor to create and modify EXEC files, and to examine and modify sequential data files written by BASIC programs. See Appendix F for instructions on editing BASIC programs. The Editor cannot work with random-access text files.

By the Way: When you type a file command to the Editor, type the pathname just as you would under ProDOS. The pathname is required when you refer to a file or a volume, no matter where you are in the Editor/Assembler; you cannot refer to a volume by its slot-drive location.

Loading a Text File

To load a text file that is stored on disk, type

LOaD

and the pathname, and press RETURN. The Editor then loads the specified text file into the Editor's text buffer. Any text previously in the buffer is erased! (If you use a partial pathname, the ProDOS prefix will be applied to your pathname.) You can then use the Editor's commands to modify the file.

If the Editor does not find the specified file, it displays the message FILE NOT FOUND. If you try to load a null file, you will see the message PRODOS ERROR = $4C. Other error messages that may appear are discussed in Appendix D.

Combining Two Files Into One

You can combine two or more text files by using the Editor's Append command.

To append a text file to the end of the text that you are currently editing, type

APPEND <pathname>

where <pathname> is the pathname to a file on a mounted disk, and press RETURN. The Editor then loads the specified text file from disk into the text buffer and appends it to the end of the text already in the buffer. This command does not affect the Editor's current pathname, which remains the pathname used in the previous LOAD or SAVE command.

You can append a text file at a particular line in your program, replacing that line and any text that currently follows it with the text from the appended file. To do this, type

APPEND <line number> <pathname>

and press RETURN. The Editor first deletes the specified line number and all the lines that follow it in the text buffer, and then loads the specified text file into the text buffer, following your existing text.

By the Way: To insert one text file into the middle of the text you are editing, follow these steps:

- First, APPEND the text file to the end of your existing text, as described above.
- Then use the COpy and Delete commands to move the text wherever you want within the text buffer. The Copy and Delete commands are discussed in the section "Manipulating Lines in the Text Buffer," later in this chapter.

Saving Your Edited Files

Don't get so busy editing that you forget to back up your work. You should **save** your text occasionally (perhaps every fifteen or twenty minutes) to minimize the amount of work you might lose due to a power failure. Although the chances of accidentally losing data are slight, a one-second loss of power to the computer can wipe out hours of editing. It is possible to lose your only copy of a file, or an entire disk, if a

power loss occurs during a SAVE to your disk. You can protect yourself against this type of loss by alternately saving your program to two (or more) disks.

Of course, you must save your text one last time when you are finished editing it.

To save the contents of the text buffer, type

SaVE <pathname>

and press RETURN. The Editor writes the contents of the text buffer into a text file, using the pathname that you specify. You can also specify a range of lines to be saved, if you don't want to save the entire file.

If you do not specify a pathname the Editor will save the text buffer using its current pathname settings. For example, if you are saving a file with the same pathname you used with the most recent LOAD command, you need only type

SAVE

and press RETURN, and the Editor will save the text buffer, replacing the older version file on disk.

You can find out the pathname that you used with the most recent LOAD or SAVE command at any time by using the FILE command.

If you have not yet specified a pathname during the current edit session (by using a LOAD or SAVE command), the Editor displays the file size parameters as if you had entered a FILE command.

WARNING
The SAVE command causes the Editor to write over any text file on the specified disk that has the same name as the file that you are saving. The Editor does not warn you that this is happening. To protect your files from being accidentally overwritten, you should use the LOCK command to write-protect the text files on a disk when you use the Editor/Assembler. If you have LOCKed all the text files on a disk and UNLOCK only the file you wish to edit, you cannot accidentally overwrite a file just by misspelling a pathname when typing the SAVE command (this can easily happen when you are working with multiple source files whose names differ only in the last few characters).

If the specified disk is write-protected, or if no space remains for your file, the Editor will display an error message. Error messages are discussed in Appendix D.

Manipulating Lines in the Text Buffer

The Editor accepts a number of commands that add, delete, copy, insert, and replace lines within the text buffer.

Adding Lines

Use the ADD command to add new lines to the end of the text buffer, or to add lines after a specified line number. To add lines to the text buffer, type

```
ADD
```

and a line number, and press RETURN.

If you simply type A, AD, or ADD in response to the command prompt and do not specify a line number, the Editor will display the number of the next line to be added to the end of the text buffer and will place you in the Editor's Input mode.

Once you are in Input mode, the Editor will accept anything that you type and will insert this text into the text buffer. You may type any number of lines into the text buffer, terminating each line by pressing RETURN. Each time you press RETURN, the Editor prompts you with the line number of the next line to be added.

When you have finished typing your last line of text, press RETURN to enter this last line, and then press RETURN again after the Editor displays the next line number on the screen. This terminates the Editor's Input mode and returns you to the Editor Command Level (the Editor displays the command prompt on a new line). The editor also terminates if you type CONTROL-D and then press RETURN. (This latter method is included for those familiar with the DOS 3.3 version of the Editor/Assembler.)

You may also use the ADD command with a line number. When you do this, the Editor places you in Input mode, as described above, but any lines that you type will be added just after the line with the relative line number that you specified. The Insert command discussed below works similarly, but Insert places any lines that you type before the line with the specified relative line number.

When you are in Input mode, you may use all of the normal Apple II input editing functions, including the arrow keys. If your Apple has the Autostart ROM, you can also use the additional ESCAPE features to move the cursor.

The Editor's Input mode allows you to place control characters into your text file. When you type control characters onto the input line and then backspace over them with the LEFT ARROW key, the Editor correctly avoids moving the cursor back across the screen. You should be careful not to insert control characters into your assembly-language programs, because the Assembler does not accept control characters. When you are using Input mode, control characters in your text are not displayed, but they are displayed (in inverse video) when you use the List command to list your text.

Inserting Lines

To insert lines into the text buffer before a particular number, type

```
INSERT
```

and a relative line number, and press RETURN. The Editor places you in the Editor's Input mode, described earlier in relation to the Add command. Any subsequent lines that you type are inserted into the text buffer, just before the line with the specified relative line number. If you type

```
INSERT 1
```

subsequent lines are inserted before the first existing line in the text file.

To terminate Input mode, type a null line. Remember that when you insert lines into your text, the relative line numbers of all lines that follow these lines in the text buffer will be increased. Check the new relative line numbers before you edit these lines.

Deleting Lines From the Buffer

To delete lines from the text buffer, type

```
DEL <first line number>-<last line number>
```

and press RETURN. First and last line numbers (separated by a hyphen) represent the first and last lines to be deleted. For example, to delete lines 10 through 20, type

```
DEL 10-20
```

and press RETURN. If you omit the hyphen and last line number, only the one line will be deleted.

WARNING
After you use the DEL command, the relative line numbers of all the subsequent lines in the text buffer are changed. This makes it dangerous to delete more than one range of lines with one DEL command.

Replacing Lines in the Text Buffer

To replace several lines of text in the text buffer, type

REPLACE <first line number>-<last line number>

and press RETURN. The first and last line numbers (separated by a hyphen) represent the first and last lines to be replaced. Using this command is exactly like typing a DEL command, followed by an INSERT command starting at the first line number specified. The Editor first deletes all lines from the first to the last number specified, then places you in Input mode, allowing you to insert any number of lines to replace those deleted. You can insert lines exactly as you would after typing an INSERT command, and you can terminate input with just a RETURN.

Copying and Moving Lines

To copy a line or series of lines from one part of the text buffer to another, type

COpy <first line>-<last line> TO <destination>

and press RETURN.

First line and last line (separated by a hyphen) represent the first and last lines to be copied. <Destination> represents the line number before which the copied lines are to be inserted. <Destination> must not fall between <first line> and <last line>. If you omit <last line>, only <first line> is copied. If used, <last line> must be greater than <first line>. The word TO is required.

To move lines from one part of the text buffer to another, you must use two commands. First, COpy the original lines to their new location, then use the DEL command to delete the lines from their original location.

Note: Remember that when you COpy lines to a new location, the relative line numbers of all text lines following that location will change. This means that when you COpy lines to a <destination> that is less than <first line>, the relative line numbers of the original lines will be changed when the Copied lines are inserted before them. If you want to delete the original lines, you must check their new relative line numbers before deleting them.

Clearing the Text Buffer

To clear all of your text from the text buffer, type

```
NEW
```

and press RETURN. The Editor clears its current text buffer, removing your work from the buffer. The current pathname is also cleared when the NEW command is used. You can use this command to clear the text buffer before typing a new text file, or in conjunction with the SWAP command described below.

To save the text in the text buffer, use the Editor's SAVE command. When you type the NEW command, the Editor does not prompt you to save your work before it clears the text buffer.

If the text buffer contains non-text data, loaded with the XLOAD command (described later), the non-text header data is cleared and the text buffer is returned to the "text" buffer state.

By the Way: If you should accidentally clear a text buffer, you can restore the buffer by using the Monitor commands. The NEW command merely resets an end-of-text pointer and does not destroy the contents of the buffer. Using the Monitor, you may be able to determine the previous end-of-text address and then set the end-of-text pointer to this address. The three text buffer pointers in the Editor are defined in the Appendixes.

Viewing Your Text in the Text Buffer

You can view the text in the text buffer either

- With relative line numbers, using the List command (control characters are displayed in inverse video);
- Without relative line numbers, using the Print command (control characters displayed as control characters).

Listing Lines of Text

To display lines of text from the text buffer, together with each line's relative line number, use the List command. It has several variations.

- To list the entire text buffer, type

  List

  and press RETURN.

- To begin the listing with a specific line other than the first line, type

  List <starting line number>-

  and press RETURN. If <starting line number> is specified without a hyphen, only that one line is listed.

- To list a range of lines, type

  List <first line>-<last line>

  and press RETURN. <First line> and <last line> must be separated by a hyphen. <First line> must be a smaller number than <last line>.

- To list two or more ranges of lines, type

  List <first line>-<last line>,<first line>-<last line>

  and press RETURN. The ranges must be separated by commas. The Editor inserts blank lines between the ranges as they are listed.

- To list a specified number of lines, starting at a specified line number, type

  List <first line>-<line count>

and press RETURN. <line count> must be smaller than <first line>, otherwise the Editor will recognize it as a <last line>. The listing will start <first line> and continue for <line count> lines.

To interrupt the listing at any point, press the SPACE bar. To continue the listing after interrupting it, press any key except CONTROL-C. If you press the SPACE bar again, the Editor will display one line and stop again. Thus you can step through the text buffer, examining one line at a time. You can cancel the listing at any time by typing CONTROL-C; the Editor returns you to the Command Level.

The List command causes the Editor to display control characters as inverse-video characters on the screen. If your system cannot display lowercase letters, they are displayed instead as punctuation symbols.

Repeating a List Command

The Editor always remembers the line numbers specified with the previous List command. To repeat the previous listing, using the relative line numbers used with the previous List command, press CONTROL-R and then press RETURN. The Editor displays the lines of text as if you had completely retyped the previous List command. Note however, that if you have added or removed lines of text since the previous List command, the content of the listed lines will be different from before.

Printing Lines of Text

To display lines of text without relative line numbers, type

Print <first line>-<last line>

and press RETURN. The Editor displays the specified text, just as with the List command, but without line numbers. The control characters that the Editor displays in inverse video when you use the List command are sent to the screen as control characters. This operation allows you to insert control characters into a file to activate any screen-control features that you may have in your 80-column text card.

By the Way: You can print or list your edit buffer to a printer using the PTRON and PTROFF commands, which are described in the section "Using a Printer With the Editor."

Viewing a Text File From Disk

You can view the contents of a text file directly from the disk, without loading it into the edit buffer--even while you have another file in the edit buffer(s).

This allows you to view a text file that is too large to fit in the edit buffer, such as a listing file created by the Assembler. This command does not format the text lines or print relative line numbers; its primary purpose is to print large listing files to the printer, as described in the section "Using a Printer With the Editor."

To view the contents of a text file (called an ASCII file by Pascal users), type

```
TYPE <pathname>
```

and press RETURN. Lines and control characters from the specified file are sent directly to the display screen. As with the LIST command, you can use the SPACE bar to interrupt or single-step the listing, or use CONTROL-C to cancel the listing.

The TYPE command does not change the current contents of the edit buffer(s).

Changing Text Within a Line

This section discusses the Find, Change, and Edit commands, which you use to find and revise text lines that are already in the text buffer.

Searching Text

To search through the entire text buffer for a particular word or character string, type

```
Find .string.
```

where string is the word or character string that you want the Editor to search for. This search string must be enclosed by two punctuation characters that do not also occur in the string itself. The example above uses periods, but any punctuation character other than dash or comma can be used. The Editor finds and lists all lines that contain the search string. The Editor lists each line no more than once, no matter how many times the search string occurs within that line.

You can limit the search by specifying on which line the search is to start, or end, or both:

```
Find begin#-end# .string.
```

If you specify both a starting and ending line number, you must separate them with a hyphen. If you specify a starting line number, but do not type a hyphen, only the one line will be searched.

To specify a **wild-card** character within the search string, press CONTROL-A. A wild-card character allows you to search for any of a set of similar character strings that may differ in one or more characters. (A wild card is a character that, for the purpose of a search, matches any other character.)

For example, if you type

```
F .TEST.
```

and press CONTROL-A, the Editor searches the entire text and lists any lines that start with TEST and end with any other character: TEST1, TESTX, etc.

Search and Replace

To substitute a new character string for some or all occurrences of an old character string in the text buffer, type:

```
Change [begin#[-end#]] .<oldstr>.<newstr>.
```

and press RETURN. The Editor searches the text buffer for occurrences of the search string <oldstr>, and replaces all occurrences with the replacement string <newstr>. You can use any punctuation character except a dash or comma in place of the periods in the above example.

As with Find, you can limit the search by specifying on which line the search is to start, or end, or both, by typing:

```
Change begin#-end# .oldstr.newstr.
```

If a starting line number is specified, but no hyphen is typed, only the one line will be searched.

When it receives a Change command, the Editor asks

```
ALL OR SOME (A/S)
```

To indicate that you wish to change all occurrences of the search string, type A. To change only some of these occurrences, type S.

If you type S, the Editor displays the line containing the first occurrence of the search string, as it would look after the change is made, and asks

```
Y/N
```

If you want the Editor to replace the original line with the changed line as it is displayed, type Y or y. If you don't want the Editor to make the change on that specific line, type N or n. In either case, the Editor then continues searching for another occurrence of the search string.

To reject the change, cancel all further changes, and return to the Editor's Command Level, press CONTROL-C.

The Editor rejects with a beep any response other than those described.

To delete a certain character string from the file, "change" it to nothing. For example, to remove all occurrences of the string JUNK from the buffer, type:

```
C.JUNK..
```

By the Way: Although the replacement string can contain "nothing," the search string must contain at least one character.

You can omit the trailing delimiting character following the new string, as the Editor also recognizes RETURN as an end of the string.

Like Find commands, Change commands can contain CONTROL-A used as a wild card in a search string. Using a wild card, you can change several similar, but not identical, character strings into the same new string.

Changing the Command Delimiter

To the Editor, the colon (:) is the command delimiter that separates Editor commands. Do not not use a colon in a search string, unless you first change the Editor's command delimiter to some other character. (The colon is not commonly used in assembly-language programs, so this is not something you'll have to do often.)

To change the Editor's command delimiter, type

```
SET Delim .xxx.
```

and press RETURN. The delimiter can be set to [,], \, ^, or _.

Entering Edit Mode

The Change command described above is useful for making identical changes to several different lines of text. To make individual changes that will be different for each line, use the Editor's Edit mode. For example, if you entered some assembler source lines without comments, you might want to go back and add individual comments to a group of lines or to all the lines containing a specific identifier.

Here are three ways to enter Edit mode.

- To enter Edit mode, with access to the entire buffer, type

  ```
  Edit
  ```

  and press RETURN.

- To edit a specific part of the buffer, specify a starting line number, or both starting and ending line numbers, type

  ```
  Edit begin#-end#
  ```

  If both starting and ending line numbers are specified, they must be separated by a hyphen.

- To edit only those lines that contain a specified search string, type

  ```
  Edit begin#-end#.<string>.
  ```

  You can specify a search string for the entire buffer or for a range of lines. You can also use CONTROL-A as a wild card in the search string.

Character Editing With Edit Mode

Once you are in Edit mode, the Editor displays the first line that you can edit, with the cursor on the first character. You can use LEFT ARROW and RIGHT ARROW to move the cursor to the left or right along the line. Using other control keys, you can replace, insert, or delete characters, seeing your changes immediately on the screen. The resulting line may be either shorter or longer than the original line.

To make changes in a text line, first use the arrow keys to place the cursor over the character that you want to change.

- To replace the character, type a new character over the old one.

- To delete a character, press CONTROL-D.

- To insert characters before the cursor position, type CONTROL-I and type the additional characters. Signal the end of the insertion by pressing an arrow key, any control character other than CONTROL-V, or RETURN.

Control Characters: If you wish to put a control character on the line you are editing, precede it with a CONTROL-V (verbatim). If you don't do this, the Editor will beep in protest.

The Editor always highlights control characters by displaying them in inverse video. Typically, you will not want to use control characters in your files, because the Assembler does not accept them in assembly-language source files.

If your system requires them, you can type lowercase characters while in Edit mode, if you enable this feature before entering Edit mode. CONTROL-E and CONTROL-W can be used to enable and disable lowercase. If you have lowercase characters in your text, they will be shown in inverse video when you place the cursor over them in Edit mode.

When you position the cursor over a tab character (normally a space), the cursor will jump over the empty space caused by the tab character. If you insert characters before a tab character, this "tabbing gap" will gradually be filled in as you type more characters. If you replace the tab character with another character, the line will jump to the left as the gap is closed.

To jump quickly around the edit line, type CONTROL-F followed by another character. The cursor then moves to the right, to the next occurrence of the specified character. (If the specified character is not found on the edit line, the cursor does not move.)

After making changes on the edit line, you may decide to restore it to its original form and try editing it again. To do this, type CONTROL-R.

If you have inserted some text and you want to save only the first part of the line that you have edited, place the cursor to the right of the text that you want to save, and press CONTROL-T (truncate). The Editor saves only the text to the left of the cursor.

If a newly displayed edit line needs no editing, or when your newly edited line is exactly as you want it, press RETURN. The Editor places the edited line back in its proper place in the text buffer, and

displays the next line for editing. When you have edited all of the lines that you specified in the Edit command, the Editor returns you to the Command Level.

To cancel Edit mode at any time and return to the Editor's command level, type CONTROL-X. The current edit line remains unchanged in the buffer.

Table 2-1 summarizes the keys and key combinations used in Edit mode.

Table 2-1. Summary of Edit Mode Keys

To Do This:	**Use This Key:**
Move cursor **left** one character	LEFT ARROW
Move cursor **right** one character	RIGHT ARROW
Delete one character	CONTROL-D
Insert characters at this position	CONTROL-I
Replace a character	any non-control character
Put **control character** in text (VERBATIM)	CONTROL-V, any character
Restore the original line	CONTROL-R
Find a specified character in the line and move the cursor there	CONTROL-F, any character
Save the line as is is on the screen	RETURN
Truncate the line at cursor and **save** in buffer	CONTROL-T
Cancel Edit mode, return to **Command Level**	CONTROL-X

Editing Two Files At Once

While editing an assembly-language source file, you may wish to check the contents of another source file, or make changes to another file, without saving and reloading your current edit file. In effect, you want to edit two files simultaneously. This is particularly important if you are working on a single program that spans multiple files.

You must first split the text buffer into two parts. If there is enough room in the two resulting buffers, you can edit two text files concurrently, alternating between the two buffers by using the Swap command.

If you are currently editing one text file, and you wish to edit a second one concurrently, type

```
SWAP
```

and press RETURN. The Editor splits the current text buffer, remembering the current pathname and file size. The portion of the buffer that contains the current text file becomes text buffer #1. Text buffer #2, consisting of whatever empty space there was in the original buffer, becomes the new current text buffer.

You can now load, edit, list, save, or perform any other editing operations on the contents of text buffer #2, while the Editor preserves text buffer #1 in memory. To return to text buffer #1, type

```
SWAP
```

and press RETURN again. The Editor makes text buffer #1 the current text buffer, preserving the contents of text buffer #2.

The Editor lets you know which text buffer you are currently editing by displaying the buffer number (either 1 or 2) before the command prompt on every command line.

If you try to use the ASM command to invoke the Assembler while you have a second text buffer active, the editor displays the message MULTI BUFFER ERROR. (The ASM command is discussed in Chapter 3.) You cannot invoke the Assembler until you deactivate text buffer #2.

To deactivate the second text buffer and delete any data in it, type

```
KILL2
```

and press RETURN. It does not matter which buffer you are editing when you do this. (To **save** the contents of this buffer first, type SAVE while editing text buffer #2.)

If there is anything in the normal "non-split" edit buffer when you type the ASM command, the Editor displays the message BUFFER ERROR. To save

the contents of the normal edit buffer, use the SAVE command before clearing the edit buffer.

As an alternative, you can transfer the contents of text buffer #2 into text buffer #1, and then deactivate the split-buffer mode.

- Type the NEW command while editing text buffer #1. This clears text buffer #1.

- Type the SWAP command to activate the contents of text buffer #2. If text buffer #1 is empty when you type the SWAP command, the Editor will deactivate the split-buffer mode and will treat the old contents of text buffer #2 as the new contents of the single remaining text buffer.

Altering the Display

The commands discussed in this section control various display features of the Editor. These commands do not alter the contents of the text buffer in any way.

Setting Tabs

When you first run the Editor program, the tabs are set for standard 65Ø2 assembly-language source files (the tab character is SPACE, and tabs are set at column 16, 22, and 36). If you are using the Editor to create other types of text files, you may want to change these parameters.

To change the tabs in the Editor, type

Tabs [tabcol [,tabcol [,tabcol [...]]]] [.<tabchar>.]

and press RETURN. You can specify up to 1Ø tab positions, in ascending order and separated by commas. If you type a delimiting character (shown as a period in the example above) followed by another single character and the delimiting character, the Editor will take this single character as the new tab character. If you type the Tabs command without any parameters, you will turn off the Editor's tabbing function.

Any character can be the tab character, but if you are editing assembly-language source files, you must use SPACE.

The Editor recognizes tab characters only in the first 4Ø columns of text. The Editor ignores tab positions beyond column 39, although the Assembler uses tab settings beyond column 4Ø to arrange its output listings.

The Editor uses the tab positions when it displays text lines in response to a List, Print, or Edit command. Tabs are not active when you are in Input mode (a space typed in Input mode appears as a single space as it is typed). When you later List this line, however, the Editor treats the space as a tab character, and it may be displayed as several blanks on the screen.

Tabs also cause lines to look different when you Print them rather than List them, as the Editor does not display a relative line number when Printing. Under most conditions, the same line will appear with six extra spaces between the first two fields when the line is Printed rather than Listed.

Using a 40- or 80-Column Display

If you have an Apple IIe with the Apple IIe 80-Column Text Card, or an Apple II or Apple II Plus with an ALS Smarterm 80-Column Text Card in slot 3, the Editor automatically displays 80 columns, uppercase and lowercase.

To switch from 80-column to 40-column display, type

```
COLumn 40
```

and press RETURN.

To switch from 40-column back to 80-column display, type

```
COLumn 80
```

and press RETURN.

If your Apple II system has no 80-column display capability, the Editor ignores these commands.

Truncating the Display

If you are using the Editor and a 40-column display, you may want to truncate the comment fields when you List or Print your source statements. This lets the statements fit on a 40-character line, improving readability.

To command the Editor to truncate comments, type

```
TRuncON
```

and press RETURN. The Editor then displays only up to the first space-semicolon sequence (the marker for the beginning of comments).

This command in no way affects the actual text in the text buffer; it only affects what the Editor displays when you type a List or Print command. This feature does not affect the operation of the Find, Change, or Edit commands.

To command the Editor to stop truncating comments, type

TRuncOFF

and press RETURN.

So that you won't accidentally lose your comments because they weren't displayed, the Editor automatically suspends truncation of comments when you are using Edit mode.

Leaving the Editor

There are three commands that allow you to leave the Editor command level and enter another Command Level within your Apple II system:

- EXIT allows you to invoke some other ProDOS command interpreter.
- END lets you easily start up (boot) another operating system environment.
- MON allows you to enter the Apple II ROM-resident monitor. MON also lets you return to the Editor Command Level, if you are very careful what you do in the Apple Monitor.

Each of these methods is described below.

WARNING
The Editor does not automatically save your work; you must use the SAVE command before you exit from the Editor if you wish to preserve the program you have been editing.

Exiting to Another Command Interpreter

After finishing with the Editor/Assembler, you can go to another ProDOS system program without having to start up again (reboot). Two versions of the EXIT command allow this:

- To exit to the standard ProDOS BASIC (named BASIC.SYSTEM), type

  EXIT

  and press RETURN. The Editor attempts to load a system file named BASIC.SYSTEM over the edit buffer, using the current prefix. After loading the file, the Editor selects 40-column display mode and invokes the new command interpreter, which then performs its normal startup sequence.

- To exit to some other ProDOS command interpreter, type

  EXIT <pathname>

  and press RETURN. The Editor attempts to load the system file with the specified pathname, over the edit buffer using the current prefix. If the file is found, the Editor selects 40-column display mode and invokes the new command interpreter. If the file is not found, the edit buffer remains intact and one of these error messages is displayed: FILE NOT FOUND, PATH NOT FOUND, DIRECTORY NOT FOUND, or BAD PATH/FILE NAME.

  The pathname can be either complete or partial, so you can invoke a new command interpreter from any online disk volume.

  To restore the "startup prefix" before exiting, type

  PFX /:EXIT <pathname>

  and press RETURN. This restores the startup prefix and then performs the EXIT command. If pathname is omitted, BASIC.SYSTEM is executed from the startup directory.

Exiting to ROM-Resident BASIC

After you finish editing a program and save it, you can return to the ROM-resident BASIC Command Level (immediate mode) by typing

END

and pressing RETURN. The Editor exits to BASIC--none of the text that you have been editing is saved. No new ProDOS command interpreter is loaded and you will not be able to execute any ProDOS commands from BASIC. The END command is normally used only to go from the ProDOS Editor/Assembler to another Apple II operating system such as DOS 3.3 or UCSD Pascal. To continue using ProDOS with another ProDOS command interpreter, refer to the EXIT command described above.

WARNING
The Editor does not automatically save your work; you must use the SAVE command before you exit from the Editor if you wish to preserve the program you have been editing.

Entering the Monitor

The MONitor command is a tool for the experienced Apple programmer. See the Apple II or Apple IIe Reference Manual for details.

To exit from the Editor and enter the Monitor, type

```
MON
```

and press RETURN. You can now use any of the Monitor commands. If you plan to return later to the Editor, do not disturb any of the memory areas (see Appendix H) used by ProDOS or the Editor/Assembler. Note also that if you enter BASIC from the Monitor, you will destroy memory areas used by the Editor/Assembler.

To return to the Editor Command Level from the Monitor, press CONTROL-Y and RETURN.

Identifying the Absolute Location of Text in Memory

When using the Apple Monitor to manipulate text from the text buffer, you may sometimes need to know the absolute location in memory of a particular text line. To find the location from the Editor command level, type

```
Where <line#>
```

and press RETURN. The Editor then displays the absolute memory address of the first character of the specified line of text.

To learn the current memory address of the beginning of the text buffer, type

```
Where 1
```

and press RETURN. The Editor will normally respond with

```
=$801
```

(equivalent to decimal 2049), the normal starting location of the Editor's text buffer.

Loading and Saving Non-Text Files

The Editor provides four commands that allow you to load non-text files (such as BINary) into the Edit buffer and then to save them with different names on different disks.

The BLOAD and BSAVE commands let you load binary files from disk into the edit buffer, and then save them from the buffer back to disk. The XLOAD and XSAVE commands let you do the same with other types of non-text files.

For example, you could move the three files of the Editor/Assembler system itself from the ProDOS Assembler Tools disk to a ProFile hard disk.

If you are not yet familiar with the concepts of binary data files, you should read the chapter on Binary Files in BASIC Programming With ProDOS.

By the Way: The Editor does not provide for editing or displaying these non-text files while they are in the buffer. Do not use LIST while in the Editor, as this displays garbage on the screen and can cause the system to hang. Instead, use the MON command to access the Apple II Monitor, which can be used to display and modify the files in memory.

Loading Binary Files

To load a binary, or BIN, file from disk to a specific address in the edit buffer, type

```
BLOAD pathname,A[$]address
```

and press RETURN. Use the optional $ only if the address is given as a hexadecimal number. Omit the $ if the address is given as a decimal number.

The address must fall within the limits of the current edit buffer, and

the entire binary file must fit between the specified address and the current end of the edit buffer. If it does not fit, you will see the message FILE TOO LARGE. Other errors that may occur are FILE TYPE MISMATCH and the four pathname errors. The BLOAD command does not support partial loading of a binary file into the edit buffer.

WARNING
The Editor does not warn you if you already have a text file in the edit buffer that might be partially or totally destroyed by BLOADing a file over the data already in the edit buffer. You should SAVE your text file(s) to disk before using the BLOAD or XLOAD commands.

You can protect the current text file from the BLOAD (but not the XLOAD) command by using the SWAP command to split the edit buffer. The BLOAD command can then load a binary file only to edit buffer #2, protecting the text in buffer #1.

Saving Binary Files

To save a binary file from the edit buffer or other memory to disk, type

```
BSAVE pathname,A[$]address,L[$]length
```

and press RETURN. The specified pathname is used to create or write over a binary file with <length> bytes of data from memory, beginning at the specified address. The binary file's load address is set into the file's directory entry from the specified address.

If you use the pathname to an existing file of some type other than binary, the error message FILE TYPE MISMATCH is displayed, nothing is saved, and the original file remains unchanged.

The BSAVE command saves from any 16-bit address or with any 16-bit length, but if the binary file already exists when the BSAVE command is given, the original file must be smaller than the unsplit edit buffer. If the original file is not smaller than the unsplit edit buffer, the FILE SIZE MISMATCH error occurs. Avoid this problem by DELETEing the existing file before BSAVEing a binary file that is larger than the edit buffer.

WARNING
Do not attempt to save the device address space from $CØØØ through $CFFF. Even reading within this address range can activate or deactivate hardware devices, such as Disk II motors or the 8Ø-column peripheral card in use by the Editor.

For more information on the problems and dangers of this type of command, see the Introduction to BASIC Programming With ProDOS.

Loading and Saving Other Data Files

The ProDOS operating system can share with Apple III's SOS operating system numerous binary, memory image, program, code, and other types of files. The ProDOS Editor/Assembler creates three types of binary files. The Editor's XLOAD and XSAVE commands allow you to load and save most of the ProDOS and SOS file types while automatically preserving the file's specific filetype, access, and auxtype information.

The XLOAD and XSAVE commands operate correctly only on contiguous sequential files that will fit within the Editor's edit buffer. These commands do not function correctly for random-access or sparse files of any type.

WARNING
If the files you XLOAD and XSAVE are sparse files, the Editor's XLOAD and XSAVE commands will attempt to operate on this type of file and give no warning that incorrect results have been generated.

This problem is not normally significant because most sparse or random-access files have such large end-of-file mark values that they do not fit within the edit buffer. If a small sparse file is XLOADed, it is converted into a sequential file when it is XSAVED; the missing (sparse) blocks are back-filled with binary zeros.

The XLOAD command must always precede the XSAVE command. This rule is enforced by the XSAVE command, which will not operate if the proper internal control information has not been created by the XLOAD command before the XSAVE command is issued.

SAVE your text file before using XLOAD and XSAVE. The text file is destroyed if the XLOAD command reads a non-text file into the edit buffer.

To load a sequential file into the edit buffer, type

```
XLOAD Pathname[,A[$]address]
```

and press RETURN. The Editor attempts to find a file with the specified pathname and examine its directory entry. If the entire file plus the directory information can fit within the edit buffer, the entire file and its filetype, access, and auxtype data are loaded. Otherwise, the error message FILE TOO LARGE appears.

If the load address (optional) is specified, the file data is loaded, starting at that address. The directory information is always placed at the beginning of the edit buffer.

The XLOAD command cannot read the four file types UNK, BAD, TXT, and DIR. If such a read is attempted, the message FILE TYPE MISMATCH appears. The LOAD and SAVE commands can be used for ordinary TXT files.

The four directory items that are saved along with the file's data are placed near the beginning of the edit buffer, just before the beginning of the data. The exact contents of these four data items can be displayed with the FILE command.

When you load a file into the buffer with the XLOAD command, the first two bytes of the edit buffer are set to binary zeros, indicating that the edit buffer contains non-text data. The FILE command examines these two bytes; when they are zero it interprets the next four bytes as the file's FILETYPE, ACCESS, and AUXTYPE. (The file consists of a six-byte header followed by data.) FILE displays three extra lines with this information after an XLOAD command. The NEW and LOAD commands clear the two zero flag-bytes created by the XLOAD command.

To save a sequential file from the edit buffer, type

```
XSAVE pathname [,A[$]address [,L[$]length] ]
```

and press RETURN. The Editor attempts to save the specified sequential file from the edit buffer. The file's length is determined by the current edit buffer size, set by the previous XLOAD command, unless the optional address and length are specified. The address specifies the beginning of the memory area to be saved. If either length or address is specified, the other must also be specified.

WARNING
If the specified pathname refers to a file of the same name that already exists, but has a different file type, the message FILE TYPE MISMATCH appears. If a file of the same type already exists, it is deleted and a new one is created with the same name. No warning is given that this is happening.

The directory information saved by XLOAD is used to create the directory entry of the new file. The XSAVE command does not operate if the edit buffer contains a text file; the XLOAD command must be executed before the XSAVE command.

To move almost any sequential file (of 37K or less) of any filetype, type

XLOAD <pathname>:XSAVE <pathname>

and press RETURN. Assuming the two pathnames are different, or have different volume names, you can move a file without knowing anything more about it than that it is sequential. The following are all sequential files that can be moved in this manner: all UCSD Pascal code files; Business BASIC, Applesoft BASIC, Integer BASIC, binary, and relocatable program files; UCSD .SYSTEM files, the ProDOS SYS interpreters, and SOS system files.

Managing Disk Directories

There are five directory management commands that let you manipulate the files within root directories and subdirectories and create new subdirectories on disk. These commands are similar to those with the same names in ProDOS BASIC.

Creating a New Subdirectory

The purpose of the CREATE command is to create a subdirectory file within which you can place other files. You cannot use this command to create any other type of file. The SAVE, BSAVE, and XSAVE commands create other types of files.

If you try to save files into a subdirectory that you have not yet created, the message PATH NOT FOUND appears, but the data in the edit buffer remains intact while you create the subdirectory and again save the file.

To create a new subdirectory file, type

CREATE <pathname>

and press RETURN. A new subdirectory file is created and given the specified pathname. This subdirectory initially holds up to twelve files, but ProDOS will automatically extend the directory to accommodate more files as you add them.

Renaming Files and Volumes

To change the name of an existing file or volume, type

RENAME <oldpathname,newpathname>

and press RETURN. The name is changed from <oldpathname> to <newpathname>. The new name must be in the same directory as the old name. For moving a file from one directory to another, use the Editor's LOAD and SAVE commands.

You can also use this command to rename a root directory or volume name. For example, you could rename your ProFile™ rigid disk volume by typing

RENAME /PROFILE,/PRO

You can also rename a subdirectory file.

You cannot rename a file that is locked. See Locking Files and Unlocking Files, below.

Deleting Files

To permanently remove a file from a disk directory, type

DELETE <pathname>

and press RETURN. ProDOS deletes the specified file and frees its disk space for use by other files. The command must include the pathname to an existing file; otherwise the message FILE NOT FOUND appears.

Locking Files

At times you will want to protect individual files from being accidentally renamed, deleted, or altered. To lock a file, type

LOCK <pathname>

and press RETURN. This command changes the status flags in the

directory entry for the specified pathname, preventing any changes to the file until it is unlocked. When a file is locked, an asterisk appears to the left of its filename in the catalog display.

Unlocking Files

Before deleting, renaming, or otherwise changing a locked file, you must first unlock it by typing

```
UNLOCK <pathname>
```

and pressing RETURN. Any file can be unlocked except a volume directory file. When a file is locked, an asterisk appears to the left of its filename in the catalog display.

Using a Printer With the Editor

Three Editor commands allow you to direct the Editor's normal screen output to a printer instead. You can print a CATALOG listing, search the file for all occurrences of a string and print all the matching lines, or print all of the text lines from the buffer. (The EDIT command will not function correctly if the printer is on.)

First, use the PR# command (see below) to define the printer interface card slot and optional printer initialization characters. Then use the PTRON command to activate the printer.

After each Editor command line is typed from the keyboard, the printer control flag, set by PTRON, causes the printer to be enabled in place of the screen display. During execution of the command, all output normally directed to the screen display is sent to the printer.

Setting Up the Printer

Before you use the printing commands, you must define the slot in which the printer interface card resides, and define any initialization characters your printer or its interface firmware may require.

To define the printer's slot, type

```
PR# n
```

and press RETURN. n represents the number (from 1 to 7) of the slot that contains an intelligent printer interface card attached to a printer. Printer interface cards are generally placed in slot 1.

If your printer and its interface firmware don't require any special initialization, this is all you need to do before using the various Editor commands.

If your printer or its interface firmware needs to be preset to a special state before printing begins, you can define a string of up to 31 control or printable characters that will be sent to the printer before the text file is printed.

To define the printer's slot and initialization characters, type

PR# n,<printer-init-string>

and press RETURN. n represents the printer slot number. After the comma are any characters (including control characters but not ESC) required to initialize the printer.

For example, if you have an Apple Parallel Interface Card in slot 1 and do not wish to echo the printed characters to the 40-column screen during printing, you would type

PR# 1,*N

and press RETURN. * represents a CONTROL-I character. This command causes the Editor to send a CONTROL-I and the letter N to the printer before any text is sent, turning off the 40-column screen echo of the printed text. In addition, if your printer has formfeed control, you could follow the letter N with a CONTROL-L, to ensure that your printout begins at the top of a new page.

Activating the Printer

Once you have defined the printer slot with the PR# command, you must then enable the printer output mode. Nothing happens when you enter this command, except that the printer control flag is turned on.

The Editor Command Level restores the console display device after executing each command. After each command is read in (either from the keyboard, the command stack, or an Exec file), the printer control flag, when it is on, activates the printer and executes the command. Any Editor command, except EDIT, can send its output to the printer.

To enable the printer, type

PTRON

and press RETURN. This command can be included with other commands in the same command line, separated by the command delimiter (normally a colon).

To disable printer output mode, use the PTROFF command. For example,

```
PTRON:CATALOG:PTROFF
```

activates the printer, prints the 8Ø-column format directory listing, and deactivates the printer.

Printing Text Files

When there is a text file in the edit buffer and you have used the PR# command to set up the printer, you can print all or some of the text lines, without line numbers.

To print all the lines in the buffer, type

```
PTRON:PRINT:PTROFF
```

and press RETURN.

To begin and end printing at specified lines, type

```
PTRON:PRINT <start>-<end>:PTROFF
```

where <start> and <end>, separated by a hyphen, represent the first and last lines to be printed. If -<end> is omitted, only one line will be printed.

To print from a specified line to the end, type

```
PTRON:PRINT <start>-:PTROFF
```

To print from the beginning to a specified line, type

```
PTRON:PRINT -<end>:PTROFF
```

The PRINT command prints the text lines with the tabs expanded according to the current tab settings. Any control characters in the text lines will be sent to the printer as control characters, possibly causing various actions by the printer.

Note: To include the Editor's relative line numbers along with the text lines, substitute LIST for PRINT in the commands above. The lines in the text buffer are sent to the printer in exactly the same way as the LIST command would display them on the screen, except that control characters appear as capital letters or punctuation characters.

To terminate printing at any time, type CONTROL-C. To single-step or pause the printing, just as you would if you were listing on the screen, use the SPACE bar. To resume full-speed printing, press any key other than CONTROL-C.

Printing Text Files Directly From Disk

To print the contents of a text file on disk, without first putting it into the edit buffer, type

```
PTRON:TYPE <pathname>:PTROFF
```

and press RETURN. The file is read from the disk file, one line at a time, and sent to the printer. To cause the printing to pause, press SPACE. To cancel the printing, press CONTROL-C.

The primary purpose of the TYPE command is to print large listing files generated on disk by the Assembler. The TYPE command prints exactly what is in the text file, including control characters, without tab expansion.

Automatic Command Execution

The EXEC command is used to execute an Exec file. An Exec file is a sequential text file that consists entirely of commands to the Editor. While the Exec file is executing, the Editor takes its commands from the Exec file instead of from the keyboard. Exec files can contain any of the Editor commands that you can type. They cannot contain Input mode text lines or Edit command control or input characters.

The various uses of EXECs are not always obvious, but the most common use is for setting up standard command sequences for invoking the Assembler. Multiple assemblies can easily be executed in sequence, for construction of complex software products, such as this 4-section 2-overlay Editor/Assembler.

EXECs are also useful for moving groups of files from one directory or disk to another. Since an Exec can enter Input mode or Edit mode and temporarily accept keyboard input that is placed in the edit buffer, you can create some very useful sequences.

Because Exec files are externally identical to all other Editor text files, you might want to name them to make them stand out in your directories. One way is to begin all Exec filenames with the letter X. Another is to end them all with letters ".EXEC".

Creating Exec Files

You can use the Editor to create Exec files, just as you use it to create text files. To create a new Exec file, perform these three steps:

- Save anything you might be editing.
- Use the NEW command to clear the buffer.
- Use the Add command to enter Input mode.

From Input mode, you can enter lines of text that contain Editor commands. You can include LOAD, SAVE, ASM, RENAME, DELETE, BLOAD, BSAVE, or almost any other Editor command. When you have created your file of Editor commands, terminate Input mode in the usual way and then SAVE the Exec file to disk. You can LIST, PRINT, change, or edit the Exec just as you can any other text file.

Do not use the CREATE command in an Exec file, because it does not work correctly on a repeated basis. Do not use the END, EXIT, or MON commands, except at the end of an Exec. Returning from the Apple II monitor to the Editor terminates an active Exec.

An Exec file can contain the EXEC command once. It must be the last command in the file, because when it is executed, the Exec file it came from will be closed and the new one opened. Avoid a loop in which an Exec file calls itself--it will never terminate as long as all the commands can be successfully executed.

If you put an Add, Insert, or Edit command in an Exec, the Exec will stop reading from the Exec until you terminate Input mode or Edit mode in the usual way, from the keyboard. Once you do this, the Exec resumes automatic command execution from the Exec file.

Note: Whenever an Editor command error occurs, the active Exec file is terminated. Subsequent commands in the Exec are not executed. This termination prevents attempts to continue execution in spite of a missing file, misspelled pathname, or other problem.

Executing Exec Files

The EXEC command causes the Editor to take commands from a sequential text file instead of from the keyboard. To invoke an Exec file, type

EXEC <pathname>

and press RETURN. If the file having the specified pathname is not a text file, the message FILE TYPE MISMATCH appears.

WARNING
Before using an Exec, be sure you know exactly what it is going to do. Invoking an Exec file alters the contents of the edit buffer if the Exec contains commands that modify the edit buffer. The system does not warn you that you have a text file already in the edit buffer that may be destroyed or modified by the commands executed from within the Exec file.

Editor execs are always executed from the beginning. The progress of an Exec can be monitored during execution. As the commands are read, they are echoed on the console display, preceded in column 1 by a +.

WARNING
After an Exec file has been started, there is no direct way to stop it. You can remove, at your own risk, some resource, causing an error that will terminate the Exec. Opening the door of a disk drive can serve this purpose, but be certain that the Exec is not trying to write on the disk at the time.

The EDASM.AUTOST Startup Exec

After loading the EDASM.ED file and initializing for normal operation, the EDASM command interpreter (EDASM) searches the startup volume directory for a text file named EDASM.AUTOST. If a text file with this name is found, it is assumed to be an EDASM Exec file containing EDASM commands. The presence of the file in the directory causes the automatic invocation of this Exec as if you had typed the command

```
EXEC EDASM.AUTOST
```

and pressed RETURN. If this file is not found in the startup directory, the first command is requested from the keyboard in the normal manner. When the commands from the AUTOST Exec are completed, Command Level input reverts to the keyboard.

Using Execs With the Assembler

You can use an Exec file to invoke the Assembler. This procedure commonly consists of clearing the edit buffer, setting the tabs for the assembly listing, selecting the printer slot or disk listing output, then invoking the ASM command, combining multiple object files into executable modules, and possibly sending a listing file from disk to printer.

The ASM command must not share a text line with another command. This is true whether the ASM command is entered from the keyboard or from an Exec file. The return path from the Assembler to the Editor clears the multiple command flag and displays the current ProDOS date and time on the console. This provides an accurate method of timing the speed of the Assembler.

Chapter 3

The 65Ø2 Assembler

Chapter 3

The 65Ø2 Assembler

Chapter 3

The 65Ø2 Assembler

About This Chapter

The 65Ø2 Assembler in the ProDOS Assembler Tools enables you to program your Apple II in assembly language.

You used the Editor to create assembly-language source files. Before these can run on your Apple II, they must be translated into executable object programs. That is where the Assembler comes in. The Assembler can also help you to locate problems in your programs, by generating assembly listings, error summaries, and symbol table listings to accompany your programs.

This chapter is organized the same way as Chapter 2. It consists of three parts:

- An overview of the Assembler.
- A brief tutorial on the use of the Assembler. This tutorial takes up where the one in Chapter 2 left off, leading you through an assembly of the program source file that you created.
- Four detailed reference sections on the Assembler's features and statements: Using the Assembler, Assembly Language Source Files, Giving Directions to the Assembler, and Using Macros in Assembly-Language Programs.

In addition, Appendix B is a summary of the 65Ø2 assembly-language mnemonics and addressing syntax, and the Assembler directives that you may use in writing your assembly-language programs.

Reminder: This chapter tells how to use the Assembler, but does not teach how to program in 65Ø2 assembly language. Several manuals on 65Ø2 programming are listed in the Preface, and you should be familiar with one of these books before you continue reading this chapter.

Overview

The Assembler creates an executable object program by translating each of the assembly-language statements in your program source file into an executable opcode (machine-language operation code). Your assembly-language statements can include any of the standard 65Ø2 assembly-language mnemonics and addressing syntax.

Your source file can also contain Assembler directives, or instructions to the Assembler itself. You can assemble very large 65Ø2 programs by chaining several source files together. These features are described in this chapter.

The Assembler can create both Binary object programs and Relocatable object programs.

- Binary programs can be run directly using the ProDOS BRUN command. You can also generate ProDOS system programs that can be executed with the ProDOS dash command.

- Relocatable programs are assembly-language programs that you can load and run during the execution of a BASIC program. Relocatable object programs are discussed later in this chapter and in Chapter 5.

Tutorial

If you follow the steps in this brief tutorial, you will

- Assemble the assembly-language source file you created in the previous tutorial (Chapter 2)
- Store the resulting binary program on disk.

To use this tutorial, you need

- Your Apple II computer running the Editor/Assembler program
- The TESTPROGRAM text file that you created during the Editor tutorial in the previous chapter.

Note: Chapter 4 contains a Tutorial that shows how to use the Bugbyter program. To do that, you'll need an executable binary program. It is recommended that you follow the steps in this tutorial, if only to prepare the necessary file for use with the later tutorial.

Getting Started

The Assembler is an integral part of the combined Editor/Assembler program. You can invoke the Assembler only from the Editor command level, but the Assembler is not resident in the computer's memory until you need to use it.

If your Apple II is not yet running the Editor/Assembler, follow the steps in the Getting Started section in the tutorial in Chapter 2. This tells how to run the Editor/Assembler and how to use the Editor command level. Once you see the colon (:) prompt character at the left margin of your screen, you are at the Editor Command Level.

To make sure that your assembly-language source program is on the ProDOS Assembler Tools disk:

1. Type the Editor's Catalog command

   ```
   CAT
   ```

   and press RETURN.

The Editor displays the directory of the current disk, which should include your program as a Text file named TESTPROGRAM.

If Your Program is Not in the Directory: Be sure you are reading the contents of the Assembler Tools disk. If your program is not on this disk, you may not have completed the tutorial in Chapter 2. Return to Chapter 2 and complete the tutorial, creating an assembly-language program and storing it on the disk. Once you have done that, return to this tutorial to assemble your program.

Assembling Your Program

Because the Assembler overwrites the Editor's text buffer as it assembles your program, you must save any edited text and clear the Editor's text buffer before you use the Assembler. Do this by using the SAVE and NEW commands, described in Chapter 2.

Note: If you do not clear the Editor's text buffer before invoking the Assembler, the Assembler displays the message BUFFER ERROR.

---

1. From the Editor Command Level (after the colon prompt), type

   ASM TESTPROGRAM

   and press RETURN.

   The Assembler program first reads your program source file (TESTPROGRAM) from the current disk, then assembles it, producing a Binary file with the name TESTPROGRAM.Ø.

   The Assembler also generates a complete assembly listing of your program on your Apple II video screen. This assembly listing should look like this:

```
SOURCE   FILE: TESTPROGRAM
----- NEXT OBJECT FILE NAME IS TESTPROGRAM.Ø
1ØØØ:         1ØØØ     1            ORG    $1ØØØ
1ØØØ:AØ CØ              2 START      LDY    #$CØ
1ØØ2:A2 ØØ              3            LDX    #Ø
1ØØ4:2Ø ØD 1Ø           4 LOOP       JSR    STORE
1ØØ7:E8                  5            INX
1ØØ8:EØ Ø5               6            CPX    #5
1ØØA:DØ F8    1ØØ4       7            BNE    LOOP
1ØØC:6Ø                  8            RTS
1ØØD:C8                  9 STORE      INY
1ØØE:98                 1Ø            TYA
1ØØF:9D ØØ 11           11            STA    BUFF,X
1Ø12:6Ø                 12            RTS
1Ø13:         ØØED      13            DS     $ED,$ØØ
11ØØ:         ØØØA      14 BUFF       DS     1Ø,$ØØ

 11ØØ BUFF            1ØØ4 LOOP          ?1ØØØ START          1ØØD STORE

** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 1Ø-JAN-83 REL-Ø3
** TOTAL LINES ASSEMBLED    14
** FREE SPACE PAGE COUNT   79
```

By the Way: If your Apple II has no 8Ø-column text card, you'll see only the first 4Ø columns of this listing.

Didn't work? If you made any typing mistakes when you typed the ASM command, the Editor/Assembler flashes an error message and returns you to the Editor Command Level. Type the ASM command again. Appendix D explains any error messages that you might see.

If the Assembler found any errors in your TESTPROGRAM source file during its assembly, you probably made a typing mistake when you created this file during the Editor tutorial. Go back to the Editor tutorial now and make sure that your file

matches the text shown in the tutorial. If not, use the Editor to fix any mistakes before you try again to assemble your program.

---

2. To verify that your new binary object file has been stored on disk, type

CAT

and press RETURN.

The Editor shows the directory of your disk. Among the files on this disk you should see a Binary file with the name TESTPROGRAM.Ø.

```
/EDASM
 NAME            TYPE  BLOCKS  MODIFIED

 PRODOS           SYS      29  2Ø-OCT-83
 EDASM.SYSTEM     SYS       8  2Ø-OCT-83
 EDASM.ED         BIN      16  2Ø-OCT-83
 EDASM.ASM        BIN      28  2Ø-OCT-83
 BASIC.SYSTEM     BIN      15  2Ø-OCT-83
 TESTPROGRAM      TXT       1  dd-mmm-83
 TESTPROGRAM.Ø    BIN       1  dd-mmm-83
```

You have now finished the Tutorial on the use of the Assembler. You have learned how to use the Assembler to do three things:

- Assemble your assembly-language source file and produce an executable binary program.
- Store the binary program on disk.
- Generate an assembly listing of your program.

The TESTPROGRAM.Ø binary (BIN) file you have created will be used in the next chapter, when you learn how to test and debug your assembly-language programs.

Using the Assembler

The preceding tutorial demonstrated one simple use of the Assembler: assembling a program source file from disk and storing the resulting object program on disk. This is the most common of the Assembler's several uses. The rest of this chapter describes each of these uses in detail.

Invoking the Assembler

To invoke the Assembler from the Editor Command Level, type the ASM command (using the following syntax):

```
ASM srcpathname[,objpathname]
```

srcpathname represents the pathname to the source file, and it must be the complete or partial pathname to a valid assembly-language source file.

objpathname represents the optional pathname (either complete or partial) to the object file. This is the pathname you wish give the output object file created by the Assembler. If you use a partial pathname, the current prefix is applied in the usual manner. (The current prefix applies to a partial pathname and is discussed in detail in Chapter 2 under the heading "The Current Prefix.")

If you do not specify an objpathname, the Assembler creates one for you by appending .Ø to the end of the srcpathname. The source filename (the portion of the pathname following the last slash) must be 13 or fewer characters, rather then the normal 15. If you forget this restriction, the assembly is cancelled, and the message ASSEMBLER PARAMETER ERROR IN LINE Ø is displayed.

You must specify a complete objpathname, beginning with a volume name, if you want the object file to be stored in a directory other than the one containing the source file. The disk volume for such an objpathname must be online; if it is not, the Assembler will cancel.

Note: Before invoking the Assembler, be sure the Editor/Assembler system files are still accessible. The Editor must load the Assembler program from disk before it can assemble your programs. The Editor remembers the startup prefix and uses it to load EDASM.ASM and, after the assembly, to reload EDASM.ED.

Suppressing the Generation of the Object File

When you are assembling your program simply to get a listing or to check for errors, you can speed up the assembly process considerably by suppressing the generation of your object file on disk.

To select this option, type @ in place of the object file name when you type the ASM command. For example,

```
ASM TESTPROGRAM,@
```

causes the Assembler to assemble the TESTPROGRAM file (found via the current prefix), generating assembly and symbol table listings, but not producing an object file on disk. The Assembler assembles your source program in the normal manner but does not write the resulting object code to disk. You can therefore have the Assembler store your object code directly into memory by using @ in the ASM command in conjunction with an OBJ directive in your source program. This feature of the Assembler is discussed later in this chapter.

Error Recovery

If you make any syntax errors when you type the ASM command, the Editor/Assembler displays an error message and places you back in the Editor Command Level.

If the Assembler cannot locate the source file you specify, or some other ProDOS error occurs, the Assembler cancels the assembly, displays the message ASSEMBLY ABORTED. PRESS RETURN, returns you to the Editor Command Level, and waits for you to respond by pressing RETURN.

As the Assembler performs its assembly of your program source file, it displays individual error messages for any errors it finds. These messages appear either on your screen or on your printer, depending on where you have directed your assembly listings. (Appendix D explains the ProDOS errors that may occur during the operation of the Assembler.)

Stopping the Assembly

To stop the Assembler at any time during an assembly, press CONTROL-C. This causes the Assembler to close all open files and free any ProDOS buffers that are in use, and then return you to the Editor command level. The Assembler does not remove any of the output files that it may have generated on your output disk. To remove these files, you can use the DELETE command from the Editor Command Level.

Under some circumstances, pressing CONTROL-C to stop an assembly can inhibit the Assembler. This can occur if the Assembler is directing output to a printer that is not online, for example. If this happens,

press RESET, pause, and press RESET again. The Assembler terminates everything and returns you to the Editor Command Level.

WARNING
Do not press RESET while the light on any disk drive is lighted, indicating that the drive is being accessed. If the Assembler is writing to the disk when you press RESET, you could destroy access to your entire disk.

The ProDOS DATE and TIME

When you type the ASM command, the Editor calls ProDOS to set the date and time from your clock card, if you have one. The date and time are converted into strings and passed to the Assembler. If you have no clock card, the date you entered at startup is passed to the Assembler, along with whatever time was previously set.

The Assembler uses the data from the clock card when you use the DATE and TIME directives in your program source file. These directives are described later in this chapter, under the heading "Controlling Assembly Listings." These directives allow you to mark your object files so that you can later associate each of your printed assembly listings with a particular object file, even if you assembled the same source program several times during a single programming session. The SBTL directive allows you to title each page of your assembly listing with the same date and time.

Generating Assembly Listings

Instead of displaying assembly listings on a video monitor, you can direct them to a printer or to a disk file for printing later. You can also control what parts of your assembly are printed in your listings.

Selecting a Printer to Receive Assembly Listings

Before invoking the Assembler from the Editor command level, you may wish to direct your assembly listings to a printer or other non-disk device output device. If you don't specify otherwise, the Assembler displays listings on the video screen.

To select a printer or other device to receive your assembly listings, type the PR# command from the Editor command level. This command has the following syntax:

```
PR# slot# [, [L#] [P#] [device-control-string] ]
```

slot# specifies the interface card slot controlling the output device to which your listings will be directed during the current Editor/Assembler session. If slot# is Ø, your Assembler listings are directed to the 4Ø-column video screen.

The three optional parameters that you can specify with the PR# command are represented by L#, P#, and device-control-string. Each of these is described below.

By the Way: The Editor/Assembler stores this data and uses it for the remainder of your current session. When you begin a new session with the ProDOS Editor/Assembler, you must again specify where to direct your listings. A most convenient way to invoke the Assembler is to include this command along with the Editor's ASM command in a ProDOS EXEC file. You can then invoke the Assembler by executing the ProDOS EXEC command from the Editor command level.

If your Apple II system has an 8Ø-column text card in slot 3, the Editor automatically selects slot 3 for the video screen, so you don't need to issue a PR#3 command to receive an 8Ø-column video display. To discontinue assembly to a printer, use PR#Ø or PR#3.

The Assembler can direct your assembly listings to only one output device. It is possible that the output device you select may echo what it receives to the display screen, but this is up to the device and how you use it. The Apple printer interface cards cannot print a listing that is wider than 4Ø columns and simultaneously echo to a 4Ø-column screen.

L# The optional logical-page length parameter specifies the number of lines the Assembler should print on a page. Unless you specify otherwise, the Assembler prints 6Ø lines per page. To specify a different logical-page length, type L followed by a two-digit number.

P# The optional physical-page length parameter specifies the number of lines from one top-of-form to the next. To specify the physical-page length, type P followed by a two-digit number. If you used a SBTL directive in your source file and you do not specify a physical-page length, the Assembler outputs a form-feed character after each page.

device-control-string The optional device-control-string is any sequence of control characters needed to initialize your printing device before printing the Assembler's output listing. This device-control-string may not exceed 32 characters.

For some Apple printer interface cards, this string typically consists of the normal CONTROL-I, N sequence that turns off the display screen and initializes the printer. Hold down the CONTROL key and press I, then release both keys and press N.

For example, to direct all assembly listings generated during this session to the printer in slot 1, with 54 lines per page, and 66 lines per form, type:

```
PR#1, L54 P66 *N
```

where * represents CONTROL-I.

Selecting a Disk File to Receive Assembly Listings

With large-capacity disk devices, ProDOS supports the creation of text files large enough to contain the listing output from your assemblies. This option is not recommended for use with Apple II systems with Disk II drives only.

By the Way: This section assumes that you have read the previous section on directing a listing to a printer or other device.

To direct your assembly listing to a disk file instead of a printer or other device, you must know which slots contain disk controller interface cards. Use the PR# command with the following syntax:

```
PR# diskslot# [, [L#] [P#] [pathname] ]
```

The Assembler determines that the slot# you have selected is a diskslot#. When the specified slot is that of a disk controller card, the Assembler assumes you want a listing file created. The device-control-string is assumed to be a ProDOS pathname instead of the usual device initialization characters.

The optional logical and physical page sizes must be specified if the printer that will later be used to print the listing file requires these options. This can result in ambiguity, if a pathname starts with either the letter L and two digits or the letter P and two digits. If your listing filename begins with L or P, resolve the ambiguity by entering a complete pathname that begins with a slash (/).

The pathname you supply may be a partial pathname, with the prefix applying as usual, or a full pathname, using any online volume. The pathname you select need not reside on the diskslot# you enter in the PR# command. The diskslot# must select a mounted ProDOS disk device,

but the pathname can select that or any other online volume.

Selecting listing output to a file requires two extra buffers, reducing symbol table space by 128Ø bytes. A very large assembly (one having a FREE SPACE PAGE COUNT of less than 5 with output to a printer) cannot be directed to a file.

A listing file consists of exactly the characters that would have been sent to the printer. This includes many spaces and control characters. You can edit this file, if it is small enough to fit in the edit buffer. Attempting to LOAD a large listing results in the Editor error message FILE TOO LARGE.

Interpreting an Assembly Listing

When you direct the listing to some device other than the video screen, the Assembler assumes that at least 8Ø columns are available for the listing. If your source lines and tabs are set to print beyond column 8Ø, either your printer must wrap the lines or tolerate more than 8Ø columns of output from the Assembler.

If you use at least one SBTL directive, headers are printed on each page of listing. The header consists of a title line followed by a blank line, as shown in Figure 3-1.

Figure 3-1. A Typical Assembly Listing

```
Ø1 CMDINTRP      COMMAND INTERPRETER      18-JUN-81 #ØØØØ1Ø PAGE 2Ø

----  NEXT OBJECT FILE NAME IS OBJ
ØCØØ:          ØCØØ    79             ORG $ØCØØ
ØCØØ:          DØØØ    8Ø EDITOR      EQU $DØØØ
ØCØØ:          DØØØ    81 ASSM        EQU $DØØØ        ;Bank 2
ØCØØ:                  83 ********************************
ØCØØ:4C 15 ØC          84 COLD        JMP TEXT         ;Hard startup
ØCØ3:4C 78 ØC          85             JMP CTXT         ;SOFT START
                                        .
                                        .
                                        .
ØC15:A2 FF             94 TEXT        LDX #$FF
ØC17:9A                95             TXS              ;Force stack
ØC18:2Ø 58 FC          96             JSR HOME
ØC1B:AØ ØØ             97             LDY #Ø
ØC1D:B9 ØB 12          98 SENDBANR    LDA BANNER,Y
ØC2Ø:FØ Ø6   ØC28      99             BEQ BANREND
ØC22:2Ø ED FD         1ØØ             JSR COUT
ØC25:C8               1Ø1             INY
ØC26:DØ F5   ØC1D     1Ø2             BNE SENDBANR
                                        .
                                        .
                                        .
12Ø8:2Ø 24 3Ø         871 MINTABS     DFB 32,36,48     ;minimum tabs
12ØB:                 872 ************************************
12ØB:                 873             LST ON
12ØB:8D 8D 8D 8D      874 BANNER      DFB $8d,$8d,$8d,$8d,$8d,$8d
1211:AØ C5 C4 C9      875             ASC '          EDITOR/ASSEMBLER II'
1225:B1 B8 AD CA      876             DATE
```

The example in Figure 3-1 illustrates many of the features of the Assembler's listing file format. The listing header contains the file number at the left, followed by the first eight characters of the pathname, and the optional title. On the right are the date, time, and page number. Below the header is a blank line.

The Assembler prints each source statement, beginning with a four-digit hexadecimal number followed by a colon. This number is the value of the Assembler's program counter when that line was assembled. Line 8Ø shows the expression result field to the right of the PC field, typical of a control directive.

The Assembler displays the message NEXT OBJECT ... when it encounters an absolute ORG directive in your source program when the Assembler is directing its object output to disk storage. Lines 84 through 1Ø2 show how normal 65Ø2 code is printed, with the branch target address to the right of the branch object code.

Lines 871 through 876 show how data directives are printed, with the NOGEN option in effect. The NOGEN option suppresses printing more than one listing line for data directives that generate more than four bytes of object output. Note that lowercase is legal input for the Assembler; it is shifted to uppercase for internal use, but is printed as given in the source.

Listing TAB Control

The Assembler listing is tabulated in the same way that the lines in the Editor are displayed. The Assembler uses the Editor tab settings to format the output listing if the first tab setting is 12 or more when the ASM command is executed. It uses its minimum tab settings otherwise. The minimum settings are equivalent to Editor tabs of 15, 19, and 31. The Assembler uses only the first three of the Editor tabs. Large Editor tab settings cause listings to be shifted off the printed page.

Interrupting the Listing

To cause the Assembler to pause when displaying or printing the assembly listing, press the SPACE bar. This interrupts the Assembler output, allowing you to view portions of the assembly listing. To restart the assembly listing, type any character that is not a mode character to the Assembler. Pressing the SPACE bar when the listing is being directed to a listing file will suspend the Assembly, but nothing will be displayed on the screen. The assembly resumes when you press some key other than SPACE or ESC.

To single-step the listing, press the SPACE bar once for each line.

If you are viewing the assembly listing on a 4Ø-column video screen, you can control a 4Ø-column window within the 8Ø-column listing. To move the window one column toward the right edge of the listing, press RIGHT ARROW. To move the window one column toward the left edge of the listing press LEFT ARROW.

The arrow keys don't stop the listing, and they change the position of the window only once for each assembler statement. If your keyboard has auto-repeat keys, you can hold down an arrow key; otherwise you must press it repeatedly until you come to the 4Ø-column window you want. Error messages are always printed within the current 4Ø-column display window.

Turning Off the Assembly Listing

The usual way to control which portions of your source program are printed in the assembly listing is to use the Assembler's LST ON/OFF directive in your program source file.

You can also control the listings from the keyboard, during the assembly. Your keyboard commands override the current state of the LST option until the next LST directive in your source program or until your next command from the keyboard. Using these keyboard commands, you can examine sections of the listing you turned off from within your source program, or you can turn off the listing of sections that were to be listed in your source program.

You would generally use these commands only when you are working on large programs. For example, when you have changed a small portion of a program, you may not want to list the entire program.

- To turn the listing ON, press CONTROL-O.
- To turn the listing OFF, type CONTROL-N (N = Not listed)

By the Way: The Assembler may encounter a LST directive within the source program that counteracts your most recent CONTROL-N or CONTROL-O command. To correct this, just repeat the keyboard command.

The Symbol Table Listing

After generating an assembly listing, the Assembler normally produces a symbol table listing. The symbol table listing can be sorted in either alphabetical or symbol-value order.

The symbol table listing is optional. There are two ways to suppress it:

- Cancel the assembly by pressing CONTROL-N.
- Use the LST NOASYM,NOVSYM directive in your source program.

To produce only the symbol table listing, use the LST OFF directive at the beginning of your source file and a LST ASYM,VSYM at the end of your source program.

When the Assembler prints the symbol table, it automatically adjusts the width of the symbol table output for either a 4Ø-column screen or a printer (assumed to be 8Ø columns). The Assembler displays the table in two columns on the screen and prints it in either four or six columns on the printer, depending on the option you select with the LST directive.

Figure 3-2 is an example of a symbol table listing sorted alphabetically by symbol name.

Figure 3-2. Example of Symbol Table

```
Ø9 SYMBOL TABLE SORTED BY SYMBOL          2Ø-MAY-83 12:Ø2  PAGE 22

   6D ADPTR          6F ADTBLND        2D3E ALPHAS     24 CH
N2CØØ SYMDUMP      ØØ72 SYTYPE         2D6E SWAP    ?2DF4 SWIPEIT
 2E5F TABLEND     ?2E84 TESTLBL       ?  ØA TXTBEG     77 VAL
*2E99 XXXXXX      X  76 YYYAV        X2CØØ
```

An X, N, ?, or * in column one indicates special information about the symbol:

- X indicates an external symbol (see the EXTRN directive).
- N indicates an entry point symbol (see ENTRY directive).
- ? indicates a symbol defined but never referenced.
- * indicates an undefined symbol. (If an undefined symbol appears in the symbol table, your assembly must have generated one or more NO SUCH LABEL errors.)

The next four characters are the symbol's two- or four-digit hexadecimal value. A zero page or single-byte address is signified by two spaces before the two-digit address; a two-byte address is displayed using all four digits (even if the two leading digits are zeros).

The remaining fourteen characters are the first fourteen characters of the symbol's name. All unique symbols appear in the symbol table, although only the first fourteen characters of any symbol name are printed.

Assembly-Language Source Files

The Assembler accepts as valid input any standard ProDOS text file that you create using the Editor. A standard ProDOS text file is defined as a file of logical records, where each logical record is a sequence of ASCII characters terminated by an ASCII carriage return ($ØD). Each ASCII character must have its most significant bit (MSB) set to zero.

A source file must contain a certain amount of syntactical structure before the Assembler can assemble it into an executable 65Ø2 object program. The Assembler must be able to recognize, as a valid assembly statement, each line or logical record of the source file. The sections that follow discuss the syntax of these assembly statements.

The Syntax of Assembly Statements

Each line of an assembly-language source file is called an assembly statement. The Assembler allows only one assembly statement per source line. Assembly statements consist of either an instruction statement or an Assembler directive. Assembler directives and instruction statements follow the same general syntax.

An assembly statement consists of up to four fields, as follows:

```
[label] mnemonic/operation [operand] [comment]
```

Note that the mnemonic or operation field is the only one required; the other three are optional. The fields must be separated by single space characters. (When you use the Editor to create your program source file, the Editor uses these spaces as tab indicators to format the display of your program. Although the Editor allows you to use other characters as Editor tab characters, the Assembler recognizes only the space character to separate fields.)

This is an example of an assembly statement:

```
BEGIN LDX #64 ;load size of buffer to be cleared
```

By the Way: Programs can also contain comment lines. The Assembler considers any line with an asterisk or semicolon as its first character to be a pure comment line. The mnemonic field of a comment line must be empty.

The Label Field

The label field is optional in an assembly statement. Any assembly statement except a pure comment can have a label. You can label an assembly statement with an identifier, which should start in the first character position of the line.

An identifier is a character string that starts with a letter and contains only letters, numerals, and periods. All characters in an identifier are meaningful, except that lowercase letters are treated the same as uppercase for all purposes except printing.

An identifier is a symbolic name that represents a 16-bit numeric value. Whenever you place an identifier in the label field of an assembly statement, you define a numeric value to be associated with that identifier. This numeric value is either the memory address that is the current value of the Assembler's program counter, or the value of the operand expression if the identifier precedes an EQU directive. If you attempt to define the same identifier more than once in a program, the Assembler displays the message DUPLICATE SYMBOL.

As a practical matter, limit identifiers to eight or ten characters. Although there is no limit to the length of an identifier, long identifiers slow down assemblies, because it takes time to compare all the characters. Long identifiers require more space in the symbol table.

The one-character identifiers A, X, and Y represent registers in the 65Ø2 microprocessor. You cannot use these identifiers to refer to any other register, address, or data in your source programs. Because some 65Ø2 assemblers also reserve the use of the S and P identifiers, you should not use these identifiers if you wish to keep your source programs compatible with these other assemblers.

You can define any absolute identifiers, or 16-bit identifiers, anywhere in your program. If you want to use a "Page Zero" identifier, one that refers to an address less than 256 decimal, you must define this identifier by using either a DSECT or EQU directive before you use it in an operand expression. If you use an identifier before you define it, the Assembler assumes that the identifier refers to an absolute (2-byte) memory address, even if you later define the identifier to represent a "Page Zero" or one-byte address.

It is a good practice to define all data identifiers at the beginning of a program, using the Assembler's EQU, DSECT, and DEND directives.

The Mnemonic Field

The second field of an assembly source statement, the mnemonic or operation field, must contain a valid mnemonic. The mnemonic must be preceded by a space, even if the assembly statement contains no label.

A mnemonic consists of two or more letters followed by a space. A mnemonic represents either a 65Ø2 assembly-language instruction or an Assembler directive.

The mnemonics representing 65Ø2 assembly-language instructions are the standard MOS Technology mnemonics. The Assembler also recognizes several additional synonyms for branching instructions. Appendix B lists all assembly-language instructions recognized by the Assembler.

Assembler directives, or pseudo-operations (pseudo-ops), are instructions to the Assembler that direct the course of the Assembly, define program data, reserve space, or perform other Assembly chores. Each Assembler directive is described in the section "Giving Directions to the Assembler," later in this chapter. Appendix B lists all Assembler directives.

The Operand Field

This field is required only for some assembly instructions and Assembler directives. When you use a mnemonic for an instruction or directive that requires one or more operands, you must include these operands in the operand field, separating them with commas. There must be no spaces within or between operands.

Each operand in the operand field must be either a register; an identifier; a constant; or an expression composed of constants, identifiers, and one or more operators. Each of these is described below.

Registers In an operand field, these are the 65Ø2 microprocessor registers. You can reference these registers by using the reserved identifiers A, X, and Y.

Identifiers These are described in the section "The Label Field," above. When you use an identifier in the operand field, the Assembler evaluates the operand by replacing your identifier with the 8- or 16-bit value that the identifier represents. Somewhere in your program, either before or after you use the identifier in an operand field, you must define a value for each identifier used. If you do not define a value for an identifier, the Assembler displays the appropriate error message.

Constants In an assembly-language program, a constant defines an explicit value. You can define two different types of constants in your programs: numeric constants or string constants.

A numeric constant represents a one- or two-byte numeric value. You can represent a numeric constant in decimal, hexadecimal, binary, or octal notation:

- In decimal notation, a numeric constant is represented in base ten by a positive integer between Ø and 65535, using the digits Ø through 9. If a value greater than 65535 is used as a numeric constant, the Assembler displays a numeric overflow message.

- In hexadecimal notation, a numeric constant is represented in base sixteen by a $ followed by up to four hexadecimal digits. The hexadecimal digits are the characters Ø through 9 and the letters A through F.

- In binary notation, a numeric constant is represented in base two by a % followed by any 16-bit binary number. This binary number is composed of the digits Ø and 1. If this binary number contains more than sixteen digits, the Assembler displays a numeric overflow message.

- In octal notation, a numeric constant is represented in base eight by the @ followed by up to six octal digits. The octal digits are the numbers Ø through 7.

String constants represent sequences of ASCII characters. You can represent a string constant in your source program by enclosing the characters between single quotes. For example, 'A' and 'AE' are valid string constants. A string may be up to 24Ø characters long, but the string must be defined all on one line. When you use a string constant as the operand of an immediate-mode instruction, you need not use a trailing quote, because such an operand may be only one character long.

When a string constant is assembled, the Assembler allocates one byte for each character in the string. The lower seven bits of each byte correspond to the ASCII code for that character, and the most significant bit (MSB) is either set or not set. You can use the MSB and DCI directives to control the state of the most significant bit (see the descriptions of these directives in the section "Generating Data in Your Object Code," later in this chapter).

Expressions The Assembler recognizes and evaluates simple numeric expressions in place of a numeric constant or identifier. A numeric expression consists of constants, identifiers, and one or more operators. The Assembler recognizes four arithmetic operators and three binary-logic operators.

The Assembler recognizes the four arithmetic operators + (addition), - (subtraction), * (multiplication), and / (division) in expressions. When performing these arithmetic operations, the Assembler does not check for numeric overflow of the results; it just retains the 16-bit

results. Thus you can use these operators to perform wrap-around memory address calculations.

The Assembler recognizes three binary-logic operators to perform logical operations on two 16-bit values. The characters used are ^ (AND), | (OR), and ! (exclusive-or).

Note that these are full 16-bit operators. All of them, especially the exclusive-or operator, can produce unexpected results if applied to 8-bit constants or expressions.

High-byte and Low-byte Operators The Assembler always maintains a 16-bit value when it evaluates a numeric expression. To extract an 8-bit result from this 16-bit value, use the high-byte < and low-byte > operators to extract the high order or low order byte from a 16-bit value.

To help remember the meanings of the characters that represent these operators, think of these two characters as arrows pointing to either the left or the right half of a hexadecimal constant. For example,

```
<($FF11) is equivalent to $FF

>($FF11) is equivalent to $11
```

Expression Syntax The syntax of a valid Assembler expression, in Backus-Naur Form (BNF), is

```
Term := Constant , Identifier

Opr := + , - , * , / , ! , ^ , |

Byteopr := > , <

Expression := [Byteopr] Term [Opr Term]...]
```

This syntax definition says "An expression is a term preceded by an optional byte-operator and followed by one or more optional [operator term] sequences." The Assembler evaluates each operator, from left to right. If you do not follow this syntax when forming an expression, the Assembler will signal an assembly error.

An arithmetic or logical operator cannot be the first or last element of an expression. The only exceptions are the plus and minus operators (+ and -), which can be used to indicate a positive or negative expression.

Expressions should not contain blanks: if an expression contains a blank, the part after the blank is treated as a comment and ignored by the Assembler.

If you have directed the Assembler to generate relocatable object code (using the REL directive, discussed later in this chapter), the Assembler does not allow multiplication, division, or the binary-logic operators to be applied to a relative identifier or subexpression, because these expressions would generate a nonrelocatable result. Also, the high-byte and low-byte operators generally do not produce a correct relocatable result. This restriction applies only if you use the REL directive to generate relocatable output from the Assembler.

The Location Counter You can use the asterisk (*) in an operand expression to represent the current value of the Assembler program counter, or PC. The asterisk normally represents the value of the program counter at the beginning of the line or statement containing the asterisk.

If the asterisk follows an instruction mnemonic, the program counter points to the address of the first byte of that instruction code. If the asterisk follows either DW or DDB, both of which accept a list of items, the the program counter changes as if each item in the list were defined with a separate statement. The DFB directive updates the program counter after every four generated bytes--the number of bytes printed on each line of the listing when multiple bytes are created with a single statement.

You can perform addition or subtraction on the value of the program counter. You can also use the high-byte and low-byte operators on the value of the program counter. The high-byte and low-byte operators allow you to align code or data structures to specific positions relative to a memory page. For example, the statements

```
HEREL    EQU  >*          ;define HEREL as the low-byte of current PC

         DS   $1ØØ-HEREL ;fill to next page boundary

NEWPAGE EQU  *           ;start of next memory page
```

create an unused data area up to the next page boundary. The code or data that follows is aligned from byte zero of the next page.

The advantage of this method is that it will always produce page-aligned code, even when the size of the program preceding this code changes as revisions are made. This method can also be used to align code to other specific positions in a page, and it can be used with the ORG directive to make relative ORG adjustment to the PC.

Zero Page Addressing The Assembler normally generates a two-byte or zero page instruction whenever the operand is a properly defined zero page identifier or expression.

Sometimes, however, you may want to generate an absolute or three-byte instruction that refers to a zero page address. To get around this Assembler rule, you must equate the identifier to the desired zero page

value only after you have referenced the identifier at least once in an operand expression. The Assembler treats the identifier as an absolute identifier for the remainder of the assembly, generating absolute-addressing opcodes for all instructions using this identifier.

The Comment Field

You can use the optional comment field to document what your program is doing. The Assembler prints the comment field in its program listing, but for all other purposes the comment field is ignored.

The comment may contain any ASCII characters. It must be separated from the operand field by a space. If you want the Editor to truncate comments, begin the comment field with a semicolon (see the Editor's TRuncON command, described in Chapter 2).

WARNING
Comment fields can be no longer than 255 characters.

The Assembler also recognizes as comments statements that begin with an asterisk or a semicolon. These statements are treated entirely as comments, and though they appear in listings, they are otherwise ignored by the Assembler.

Giving Directions to the Assembler

Assembly directives are instructions that you put into your program source file. They direct the Assembler to perform certain operations during the assembly of a program.

Statements containing Assembly directives resemble those containing machine instructions in that both statements can contain label, mnemonic, operand, and comment fields. Unlike machine instructions, however, Assembly directives are not assembled into executable 65Ø2 opcodes. Only the data-definition Assembly directives generate actual data that is included in the resulting object program.

Assembly directives perform a variety of functions, including:

- Controlling the overall assembly of your program
- Assigning information

- Generating data to be included in the object file
- Controlling the conditional assembly of portions of your program
- Controlling any additional source files that will be used in the assembly
- Controlling what type of object code will be generated during the assembly
- Controlling how assembly listings will be printed.

The following sections describe each of the assembly directives that the ProDOS Assembler recognizes. Many of these directives are specific to this Assembler, although you will probably recognize similarities to other assemblers with which you may be familiar.

The discussion of each Assembly directive begins with the required syntax of the statement containing the directive. Where labels are not shown, you can include a label if you wish. Any of these statements can also include a comment.

Controlling the Overall Assembly

The directives described in this section are used to direct the overall assembly of a source program. These directives

- Control the addresses at which a program is assembled
- Permit you to generate dummy code sections to define memory locations your program can access or constants that it can use
- Control the type of object code (absolute load-image object code, absolute system program object code, or relocatable object code) that the Assembler generates.

To use these directives, include the directive mnemonic in an assembly statement, as you would any other mnemonic. Some of these directives also require that you include an operand in the statement. You can precede any of these directives with a label, and you can follow any directive with a comment.

ORG

```
ORG expression
```

The ORG (Origin) directive establishes the origin address of your object code.

WARNING
You must use at least one ORG directive in your source file, or the Assembler will not generate and save object code.

The Assembler recognizes two kinds of ORG directives: relative and absolute.

A relative ORG is one whose operand expression contains relocatable identifiers. You must have previously defined all of the identifiers that you use in the operand expression. Three examples of relative ORG directives are:

```
ORG  *+5
ORG  SYM+1Ø
ORG  *-2ØØ
```

These relative ORG directives cause the Assembler simply to update its internal program counter for the current object file. The Assembler updates this program counter both forward and backward in relation to your object file. If you increase the value of the Assembler's position counter, the Assembler fills the object file forward to the new position with random data. If you decrease the value of this position counter, the Assembler deletes any object code that it had previously generated for any addresses above the new value of the program counter.

An absolute ORG directive is one whose operand contains an absolute expression, typically a constant. You will use an absolute ORG directive to define the starting address for which the Assembler will generate object code. The Assembler also places this starting address in the output object file's directory entry, so that ProDOS can use this address when you later load the object file using ProDOS's BLOAD or BRUN command. You'll normally use just one absolute ORG directive in a program source file.

The Assembler generates a new output object file for each absolute ORG it encounters in a source file, closing the current object file (if any) and starting a new file. The new file has the same filename as the old file, except that the last character in the filename is changed to the next character in the character set

```
Ø123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

This character set is circular, with Ø following Z. For example, filename ABCDE would be changed to ABCDF; VWXYZ would be changed to VWXYØ. This limits to 36 the number of absolute ORGs in one assembly.

By using multiple absolute ORG directives in your source program, you create multiple object files. These files must be either combined after assembly, or loaded independently.

WARNING
Be sure these new filenames do not already exist as the names of unrelated files--the Assembler deletes existing output object files (BIN, SYS, or REL types) without any warning.

This design is based on the assumption that when you repeatedly assemble the same program, the old object file becomes obsolete as you make corrections in your current assembly. If this is not the case, rename or copy to another volume the older version of your object file(s).

SYS

SYS

The SYS directive changes the file type of the next object file that the assembler will create in an assembly. This directive should immediately precede an ORG statement if you are creating a ProDOS interpreter file (see the ProDOS Technical Reference Manual for details).

After the Assembler opens each object file during an assembly, it resets the object file type to binary (BIN in the catalog displays). Thus you can create a ProDOS interpreter and associated binary overlays in a single assembly, by putting the SYS directive before the proper ORG statement.

DSECT and DEND

DSECT

DEND

Use the DSECT (Dummy SECTion) directive to define an area of memory, such as a data table, or the 65Ø2 Page Zero, without actually generating any object code to reside in this memory area. The DSECT directive marks the beginning of a block or group of statements in which you define the values of identifiers that you will use elsewhere in your program. Use these identifiers to reference memory locations within the DSECT memory area. You will most commonly use the DSECT directive to

define the labels of data items and pointers in the 65Ø2 Page Zero area of memory.

The name DSECT comes from Dummy SECTion, so called because the Assembler generates no object code for this section. The DSECT directive causes the Assembler to suspend object code output temporarily, and sets the Assembler's position counter to address zero. Once you have started a DSECT memory area, the DSECT remains in force until the Assembler encounters a DEND (End DSECT) directive, described below. Within this DSECT area, you can include any assembly statements you wish, including the ORG directive.

If you include an ORG directive within a DSECT, the Assembler uses the ORG only to control the addresses that the Assembler assigns to identifiers within the DSECT. The Assembler does not treat these ORG directives as absolute ORGs, nor do these ORGs alter the Assembler's normal position counter, which is saved while the statements of the DSECT are assembled.

You cannot nest DSECT areas. That is, you cannot use a DSECT directive between another DSECT and a DEND. If you do this, the Assembler will signal a DSECT/DEND error.

The DEND directive marks the end of the DSECT statements. When the Assembler encounters a DEND directive, the Assembler resumes generating object code where it left off after encountering the DSECT directive. The Assembler restores its normal position counter address, which it saved while assembling the DSECT statements.

Here is an example of the use of the DSECT and DEND directives:

```
            .
            DSECT
            ORG     $ØØ
; DEFINE ZERO-PAGE STORAGE
AREG        DS      2
BREG        DS      2
CREG        DS      2
H           DS      1
L           DS      1
            ORG     $4ØØ
TEXTPG1     DS      $4ØØ
            DEND
            .
            .
```

Note: If you start a dummy section with the DSECT directive and never end it with a DEND directive, none of the assembly statements following the DSECT directive will appear in your object file output. These statements will appear in listings, just as in a normal assembly, but no object code will be generated. This "missing DEND" problem can be the cause of mysterious losses of object code in your output files.

OBJ

OBJ expression

The OBJ (OBJect) directive is useful only if you wish to assemble your program and place the object code directly into memory rather than storing it on disk. When you suppress the generation of an object file by typing the ASM command with the @ option for the object pathname, the Assembler generates no object output unless you use this directive to specify where in memory to put the object code.

The OBJ directive should be at the beginning of the program: only those statements that follow it will have object code generated from them.

The Assembler performs very specific checks on the expression given in the OBJ directive. You should not use a value less than the current end of the Assembler's symbol table; if you do, the Assembler displays the displaying the message OBJ BUFFER CONFLICT. In addition, you should not use a value greater than the HIMEM value passed to the Assembler by the Editor; if you do, or if the object code output exceeds this limit during an assembly, the Assembler signals an OBJ BUFFER OVERFLOW. The symbol table area normally occupies memory from $8ØØ through $7ØØØ; thus OBJ $6ØØØ allows room for 4K of object code and much less space for the symbol table.

You cannot use this OBJ directive if you use the REL directive (discussed below) to generate relocatable object code output.

REL

REL

The REL (RELocatable) directive causes the Assembler to generate relocatable object code during the assembly. The Assembler does this by appending a relocation dictionary to the end of the object code. The relocating loader program can later use the relocation dictionary to load your object program and prepare it for execution at any starting address. The relocating loader routines are described in Chapter 5.

The Assembler produces a relocation dictionary only if you include the REL directive at the beginning of your source program, before you use or define any identifiers.

WARNING
The relocation dictionary requires space within the symbol table area, reducing the space available for identifiers. An assembly that uses up most of the symbol table space without the REL option will not assemble later with REL enabled.

When you use the REL directive, your object file is given a file type of REL (RELocatable) in the ProDOS disk catalog. Relocatable object files cannot be used by the ProDOS BRUN command. This type of file can be used only with the RLOAD program described in Chapter 5.

The REL directive requires no operand. You may not use this directive in a source program containing an OBJ directive to produce coresident object code output.

When it generates relocatable object files, the Assembler clears the relocation dictionary each time it encounters an absolute ORG directive. Each segment of your assembled object code will have a separate relocation dictionary.

X65Ø2

X65Ø2

This directive is included solely for those who might want to use an Apple system to develop software for the NCR 65CØ2 microprocessor.

This directive allows you to include the nonstandard opcodes and address modes available in the 65CØ2. This microprocessor and its documentation are not available from Apple Computer, Inc. As this

manual is written, NCR's address is 2ØØ1 Danfield Ct., Ft. Collins, Colorado 8Ø525.

See Appendix B for a list of the 65CØ2's additional mnemonics.

If you use the X65Ø2 directive at least once during an assembly, before the first use of such an instruction or address mode, the Assembler accepts the 65CØ2's additional instructions and address modes and their syntax without causing the 65CØ2 ADRS MODE/OPCODE ERROR message. We leave the details of the 65CØ2 to those programmers who have the determination to acquire the necessary documentation.

PAUSE

PAUSE 1 / PAUSE 2

The PAUSE directive causes an assembly to halt and display the message PRESS RETURN TO CONTINUE and to wait for the RETURN key to be pressed once.

If the pause is to occur during pass 1 of the assembly, type

PAUSE 1

If the pause is to occur during pass 2 of the assembly, type

PAUSE 2

To request a pause at the same place in both passes, use both at the same place. This directive can be used just before a CHN or INCLUDE directive to allow programmed volume exchanges in flexible disk-based systems.

If the disk volume of the first source file is not mounted when pass one ends, there must be an extra PAUSE1 as the last statement of the last source file to allow for insertion of the proper volume before pass two begins.

WARNING

(1) Never remove a volume that is being used for output file(s), as ProDOS doesn't prevent you from writing on some other volume put in place of such an output volume. The Assembler can have two output files, the object file and the listing file, open on different volumes, so be careful how you use this directive.

(2) Don't use volume swapping unless you fully understand it. ProDOS is not designed to allow volume swapping--it provides no protection if volume swapping is used

incorrectly. Volume swapping can be used only if you swap read-only volumes. Swapping a read-only volume for a read-write volume will probably destroy the files on your disks.

(3) Volume swapping will not work in a one-drive system, because the object file must always be mounted.

Assigning Information

The directives described in this section are used to direct the overall assembly of your source program. They are used to assign numeric values to identifiers or to define the characteristics of an identifier that may reference or be referenced by another assembly-language program.

Include these directives in an assembly statement as you would any other mnemonic. Some of these directives require an operand in the statement. You can precede any of these directives with a label, and you can follow any directive with a comment.

A Note on External Symbols: You may use the DEF, ENTRY, ZDEF, EXTRN, REF, ZXTRN, and ZREF directives whether you are producing a relocatable object file or a binary object file. To correctly run a program that contains references to external identifiers, however, you must use a Linker or Linking Loader to link this module to other program modules in which all of these external references are resolved. Although no such Linker or Linking Loader is currently available, these directives are provided here for completeness, in the event that you want to write your own Linker or that a Linker or Linking Loader becomes available.

You can also make use of external symbols when you are creating self-modifying code, as a way of defining addresses that are filled in at execution time. The DEF and ZDEF directives do not cause anything to appear in the machine code portion of the object file; the EXTRN and ZXTRN directives allow undefined symbols to remain undefined without causing Assembler error messages.

EQU

identifier EQU expression

Use the EQU (EQUate) directive to define the value of a symbolic identifier. The identifier that you place in the label field cannot be defined elsewhere in your program. The Assembler evaluates the expression in the operand field and defines the identifier to have this numeric value. You cannot use any external identifiers in the operand expression.

You can use symbolic identifiers to create a source program that means something to people, rather than just to the computer. Using this directive helps you create a program that is easy to change and understand long after it is written. The EQU directive provides an easy way to name any item that has special meaning--for example, an ASCII character that might serve as a special delimiter in a program. Using this directive, rather than just a string constant in many places, makes it possible to change how a program functions without having to edit many lines.

DEF (or ENTRY)

DEF identifier / ENTRY identifier

The DEF directive signals the Assembler that an identifier defined in this module of your program may be referenced by other modules as an external symbol. You may use this directive more than once in a module, either to define global identifiers or to define alternate entry points for your module.

When you use the DEF directive in an assembly statement, the identifier in the operand field is marked as a global or entry point identifier. Normally, the value of this identifier would be defined elsewhere in your program. You may, however, define the current value of the Assembler's program counter as a global entry point by including this same identifier in the optional label field of the statement containing the DEF directive. You should not use the DEF directive inside a DSECT.

Whenever you assemble a module containing the DEF directive and select relocatable object code using the REL directive, the Assembler appends an external symbol directory (ESD) to the relocation directory (RLD) of your object code output. This external symbol directory contains all of the global identifiers and external symbols defined in your module, along with information necessary for a Linker or Linking Loader program to link this module to other modules that may reference these entry points or global identifiers.

ZDEF

```
ZDEF identifier
```

Use the ZDEF (Zero page DEFine) directive to create a zero page global identifier in the same way that you would use a DEF or ENTRY directive to create an absolute identifier.

All the rules that apply to DEF and ENTRY also apply to ZDEF, except that the identifier must be a zero page identifier. You can use the ZDEF directive only after the identifier has been defined as a zero page identifier; otherwise the Assembler displays the message ILLEGAL LABEL.

EXTRN or REF

```
EXTRN identifier / REF identifier
```

The EXTRN (EXTeRNal) directive is used to indicate that an identifier is defined externally, rather than within this module. The Assembler always treats these identifiers, or external symbols, as two-byte identifiers, never as zero page identifiers. If you use an external symbol as the operand of an instruction, the Assembler generates two zero bytes in the address portion of that instruction.

If you are using the REL directive to generate relocatable object code, and you use the EXTRN directive to define symbols as external, the Assembler adds an external symbol directory (ESD) after the relocation directory (RLD) in the relocatable object file.

To execute a program module containing external symbols, you must first use a Linker or Linking Loader program to link this module with one or more additional program modules in which these external symbols are defined. A Linker resolves all these external references, using data from the External Symbol Directories of each module, and links all the modules into a single executable object program.

You can include a label identifier in the label field of the statement containing the EXTRN directive, and this label will be defined with the current value of the Assembler's program counter. The identifier that follows the EXTRN directive in the operand field of the statement is the identifier that is defined as the external symbol.

ZXTRN or ZREF

```
ZXTRN identifier / ZREF identifier
```

This directive serves the same purpose for zero page globals that EXTRN or REF does for absolute global identifiers.

You must use this directive to define an identifier as a zero page identifier before referencing this identifier in the operand field of an assembly statement. Otherwise the Assembler displays the error message lLLEGAL LABEL.

Generating Data in Your Object Code

Use these Assembly directives to allocate or define data areas within your assembly-language program. These Assembly directives let you define

- Byte and word data
- Address tables
- Data storage areas
- ASCll character data
- Message strings.

Any of these Assembly directives can be preceded by a label and followed by a comment.

DFB or DB

```
DFB expr[,expr...] / DB expr[,expr...]
```

The DFB (Define Byte) directive is used to define one or more bytes of data to be placed in the object file. The Assembler evaluates each expression and uses the resulting value, modulo 256, as the value for each byte. lf you use an identifier in the label field of the statement containing the DFB directive, the byte generated is the low byte of the address of the identifier. lf the bytes that are generated are calculated from a relocatable expression, the Assembler makes an entry in the RLD for each such byte, so that its value can be relocated.

Separate the expressions in the operand with commas, and use no blanks between them. Any valid expression can be used in the operand field. The program counter, referenced by the asterisk pseudoidentifier, is updated for every four bytes of generated object file data.

It is better to use multiple DFB directives, limiting each DFB directive to five to ten expressions, than to use a large number of expressions in the operand of one DFB directive.

DW

```
DW expr[,expr...]
```

The DW (Define Word) directive is used to define two-byte 65Ø2 words.

In a 65Ø2 word, the lowest eight bits of the 16-bit expression are stored in the first byte, and the most significant (high) eight bits are stored in the second byte. The bytes must be in this order if the 16-bit address is to be used as an indirect address pointer in the indirect-indexed and jump-indirect instructions of the 65Ø2 microprocessor.

The label field identifier is given the value of the program counter, which is the address of the first (low order) byte of the first word. If more than two expressions are used, only the first one appears in program listings, unless you enable the LST GEN option. The program counter, referenced by the asterisk pseudoidentifier, is updated as if each expression were defined by a separate DW statement.

DDB

```
DDB expr[,expr...]
```

The DDB (Define Double Byte) directive is exactly like the DW directive, except that the bytes are stored in reverse order, with the high order byte first and the low order byte second.

The label-field identifier has the address of the first (high order) byte of the first double byte expression.

DS

```
DS expr[,expr]
```

The DS (Define Storage) directive is used to reserve a group of bytes without defining any data to be stored in those bytes. The first expression of the DS directive may contain identifiers only if those identifiers have been defined earlier in the program. This expression's maximum value is 16384 decimal.

If you include a second expression in the operand field, the reserved bytes are filled with the value of this expression. If you include only the first expression in the operand field, the reserved bytes will contain random values.

The amount of space reserved by the DS directive is included in the size of the output object module. Thus, if you accidentally enter a DS with an expression that comes up with a value of, say, 12K bytes, you will suddenly get a very large output file. To avoid wasting disk storage space, use this directive only for small data areas. Large buffers and work areas need not be stored along with your executable object program, and so should not be defined by using this directive.

If you include an optional identifier in the label field of this assembly statement, this identifier is assigned the address of the first byte of this allocated storage.

When you use a DS directive inside a DSECT, the Assembler does not actually place any code in the object file. Using DS statements within a DSECT is an easy way to define a data structure so that you can modify it later and reassemble the program without affecting other assembly statements in the source program.

MSB

```
MSB ON

MSB OFF
```

The MSB (Most Significant Bit) directive lets you control the value of the most significant bit of the ASCII characters that are generated by the Assembler. You can use the MSB directive as many times as you need to in an assembly program. The ASCII characters affected by MSB are those generated as immediate string constants, and the string operand of the ASC directive, but not the DCI directive.

WARNING
The ProDOS Assembler defaults to MSB OFF, because this is the standard definition of ASCII characters in ProDOS data files. However, the DOS 3.3 version of the Assembler defaults to MSB ON. Be careful when moving source files from the DOS to the ProDOS Assembler.

ASC

ASC .string.

The ASC (ASCII) directive defines a string of eight-bit bytes in the output object file that are filled with the ASCII values of the characters in the operand string of the ASC directive. Four or fewer bytes are printed on each source line, without a line number, if the LST Gen option is in effect for the directive. When the LST NOGen option (the default) is in effect, only one line and the first four bytes are printed. If a label is present on the ASC directive, it is assigned the current value of the program counter, which will be the address of the first character of the string constant in memory.

The operand string (shown above as .string.) is a delimited string that begins with any character that does not occur in the string, and is optionally terminated with the same delimiter. You can omit the terminating delimiter if you also omit the comment field in the assembly statement. The MSB directive controls whether the most significant bit of each character in the generated bytes is a one or a zero.

STR

STR .string.

The STR (STRing) directive is an easy way to create a string of ASCII characters preceded by a count byte.

The count byte contains the number of characters in the string, not including the count byte itself. The MSB directive controls whether the most significant bit of each character in the generated bytes is a one or a zero.

DCI

DCI .string.

The DCI directive functions just like the ASC directive, except that the MSB directive does not control the MSB of each byte.

Instead, the Assembler generates all bytes of the DCI string with a MSB of zero, except for the last byte, which will have a MSB of one.

DATE

DATE

This directive generates nine bytes of ASCII data in the output file.

The nine bytes are generated from the ProDOS date word at the beginning of each assembly. This date is converted into a sequence of ASCII characters in the DD-MMM-YY format.

TIME

TIME

This directive generates six bytes of ASCII data in the output file, from the ProDOS time word.

If the ProDOS time word contains a zero, these six bytes contain six blanks; otherwise they contain the time in the format HH:MM_, where _ represents a space.

If your computer has a ProDOS-compatible clock card installed, ProDOS is used to update the time word before each assembly, providing a unique datum in IDNUM. This information is printed in the header line of the assembly listing output; thus this directive provides a means of marking an object module so you can associate it with a particular printed assembly listing. However, you must set it externally if your system has no clock card.

Controlling Conditional Assembly

The Assembler recognizes a number of directives that let you control which sections of a program source file are included in the assembled object file. This process, called conditional assembly, is useful for writing a single program that you can later assemble to run in different environments, such as "production" versus "test," or "machine configuration x" versus "machine configuration y."

DO, IFxx, ELSE, and FIN

These nine (there are six variations of the IFxx directive) conditional-assembly directives work together to mark the beginning and end of a section of conditional assembly source statements, called a **conditional assembly block.** A conditional assembly block is a group of source statements that are assembled only if certain conditions are met.

- The DO directive marks the beginning of a conditional assembly block.

- The FIN directive marks the end of a conditional assembly block.

- The ELSE directive divides the statements between a DO and a FIN directive into two conditional assembly blocks; the second conditional block is an alternate conditional block that is assembled only if the first block is not.

The other conditional-assembly directives--IFNE, IFEQ, IFLT, IFLE, IFGT, and IFGE--are referred to in this section as the IFxx directives.

Like the DO directive, they mark the beginning of a conditional block of source statements. These directives must always be followed in the source program by the FIN directive.

Here is the format of a conditional assembly block:

```
DO expression        or    IFxx expression

    ( conditional block #1 )

  [   ELSE

    ( conditional block #2 )    ]

     FIN
```

When the Assembler encounters a DO or IFxx directive in a source program, it evaluates the operand expression and compares the result to the value zero. Any identifiers you use in this operand expression must be defined earlier in the source file.

This is how the DO and IFxx directives control the assembly of the conditional block:

- The DO and IFNE directives cause the conditional block to be assembled if the resulting expression is not equal to zero.

- The IFEQ directive causes the conditional block to be assembled if the resulting expression is equal to zero.

- The IFLT directive causes the conditional block to be assembled if the resulting expression is less than zero.

- The IFLE directive causes the conditional block to be assembled if the resulting expression is less than or equal to zero.

- The IFGT directive causes the conditional block to be assembled if the resulting expression is greater than zero.

- The IFGE directive causes the conditional block to be assembled if the resulting expression is greater than or equal to zero.

Depending on the value of the operand expression, the Assembler does one of the following:

- It assembles the statements in the conditional assembly block.

- It ignores the statements in the conditional block and creates no object code for these statements. The Assembler indicates in the assembly listing the source statements that it ignores.

To define the start of a second or alternate conditional assembly block, include an ELSE directive between the DO directive and the FIN directive. This alternate conditional block is assembled only if the first conditional block is not assembled. The ELSE directive marks the end of the first conditional block and the start of the alternate conditional block. An ELSE can be used only between a DO and a FIN.

The FIN directive marks the end of the entire conditional assembly block. When the Assembler encounters the FIN directive, it assembles all subsequent assembly statements in the source file normally.

Although the Assembler does not recognize assembly variables that you can change within your source program, you can use dummy identifiers that you define along with these conditional assembly directives to control a variety of different assembly conditions.

Here is an example of how to use the conditional assembly directives:

```
.
IFLE TBLSIZE-256     ; DOES TABLE FIT IN 1 PAGE?
DS   256             ; YES. allocate only 1
ELSE                 ; otherwise ...
DS   512             ; NO.  use two pages
FIN                  ; oh boy, all done!
.
```

FAIL

FAIL pass,.string.

Use the FAIL directive in conjunction with the conditional assembly directives to provide programmer error messages. When the Assembler encounters this directive while assembling a source file, it prints whatever message is in the string. Typically, FAIL is enclosed in a

conditional assembly block that will be assembled only if an error condition is detected.

The value of the pass expression is the Assembly pass number (1, 2, or 3) in which the Assembler will print the FAIL error message (3 means both pass 1 and pass 2). The string is the message text and, like an ASC operand, must be enclosed by delimiters.

You can use this directive to perform automatic checks for size on the matching halves of data tables or for code page alignment, and similar cross checks between things that must match in size or have particular relationships. You can use any of the conditional-assembly directives with this FAIL directive to generate these automatic warnings.

Controlling Source Files

The four source file directives (CHN, INCLUDE, SBUFSIZ, and IBUFSIZ) let you the source files that the Assembler will assemble, and control the size of the buffers that the Assembler will use to read these files.

CHN

```
CHN pathname
```

The CHN (CHaiN) directive is used to connect the segments of a large source program.

The pathname is required. Because all statements following a CHN directive are ignored, CHN should only be the last statement of a source file. If the specified pathname does not correspond to an existing file on a mounted volume, the Assembler displays the message FILE NOT FOUND and cancels the assembly.

INCLUDE

```
INCLUDE pathname
```

The INCLUDE directive is used for one level of source file nesting.

This directive causes the Assembler to suspend assembling statements from the current source file and to start reading and assembling the statements in the file specified by pathname. If the volume, path, or filename cannot be found, the appropriate error message is displayed.

The Assembler does not permit nested INCLUDE files (do not use an INCLUDE directive in an included file).

SBUFSIZ and IBUFSIZ

IBUFSIZ expression

SBUFSIZ expression

These two directives let you optimize the size of the buffers the Assembler uses when reading your source and INCLUDE files during assemblies.

The source buffer's default size is four pages or 1Ø24 bytes. The INCLUDE buffer's default size is 16 pages or 4Ø96 bytes. These defaults are close to optimal for doing multiple-file assemblies using INCLUDE files with Disk II drives.

The Assembler displays FREE SPACE PAGE COUNT at the end of the assembly listing. This count is the number of memory pages that were unused by the Assembler for symbol and/or RLD tables at the end of the assembly. You can use this information along with the IBUFSIZ and SBUFSIZ directives to increase the size of the SOURCE and INCLUDE buffers, thus increasing the speed of printing and assembly.

By the Way: If you have a Disk II drive, do not use a buffer larger than 32 pages. Doing so can result in a slower assembly time.

The Assembler evaluates the operand expression in two ways:

- If the value is less than 128, the Assembler takes the value to be the size of the buffer in pages, where a page is 256 bytes.

- If the value is greater than 256, the Assembler takes the value to be the size of the buffer in bytes. If the value is not divisible by 256, the Assembler truncates the buffer size to nearest whole number of pages.

If the resulting number of pages is greater than 127, the Assembler signals an OVERFLOW error. If the buffer size exceeds available memory, the assembly is cancelled. The assembly is also cancelled if the new size of SBUFSIZ creates a source buffer so small that the source file would be truncated.

Use these directives only within a source file. If you use these directives in an INCLUDE file, the Assembler displays the message INVALID FROM INCLUDE FILE.

Controlling Assembly Listings

These optional directives and directive options control the format and presentation of the assembly listings generated by the Assembler. They can improve the readability of assembly listings and save space in your source files.

By the Way: You can include an identifier in the label field of any assembly statement containing a listing directive, but this is not recommended. Because assembly statements containing listing directives are not printed in your assembly listings, defining identifiers in this way could result in incomplete documentation of your program.

PAGE

```
PAGE
```

When it encounters a PAGE directive, the Assembler sends an ASCII form-feed character to the output device, causing a page eject. It also sends a blank line to the video screen.

The PAGE directive itself does not print as a line on the listing, but you can detect its presence by its action and the missing line number in the listing. When you are using the Editor, you can use the missing line number to find this line.

LST

```
LST (ON , OFF) [, [NO]option [, [NO]option] ...]
```

or

```
LST [NO]option [, [NO]option] ...]
```

The LST (LiSTing) directive lets you suppress part or all of the source listing. Turning the listing off (LST OFF) can increase the speed of your assembly. This is most noticeable when you are doing a large assembly and listing it to a printer. You can use this directive any number of times to turn on and turn off selected parts your program's listing.

The eight options are specified by the first letter of the following abbreviations:

Cyc, Gen, Warn, Unasm, Asym, Vsym, Sixup, Exp

The options provide additional control of the information in the listing file. You can specify any number of options with or without the ON/OFF print control. The option processor recognizes only the first letter of each option, so you can choose your own spellings if you like. If you do not use the LST directive in your source program or do not specify any options, the Assembler assumes these defaults:

LST ON,NOCyc,NOGen,Warn,Unasm,Asym,NOVsym,NOSixup,Exp

Cyc Option If you are programming timing-sensitive code, you can use the Cyc (Cycle times) option to cause the Assembler to list the 65Ø2 instruction cycle time for each assembly instruction in the source file. The times are printed, as single digits within parentheses, to the left of the source line numbers.

NOCyc (the default) turns off the printing of cycle times.

Gen Option The Gen (Generated object code) option is used to control the printing of all object code bytes generated by the data directives, if printing this data would require more than one print line. This option affects only what the data directives generate, not what they print.

Unless you specify the Gen option, the Assembler lists only the first four bytes of object code generated by a data directive. The default is NOGen (option off), which saves printing time.

Warn Option The Warn (Warnings) option enables or disables the printing of Assembler warnings.

The default value is Warn (option on).

Unless you use the NOWarn option, the Assembler prints all warnings it encounters as it assembles your file. The Assembler prints a warning total at the end of the listing, along with the error message total, even if you have suppressed the printing of the warnings themselves.

Unasm Option Normally, the Assembler prints all statements in your source program, marking those statements that were unassembled. Use the Unasm (Unassembled source) option to suppress printing of statements in your program that were not assembled because they were part of a conditional assembly block.

The default value is Unasm (option on).

Asym Option The ASYM (Alphabetic symbol table) option produces an alphabetical symbol table dump listing. The default value is Asym

(option on). Use the NOAsym option to suppress the Assembler's normal alphabetical listing.

Vsym Option The Assembler can print the symbol table by order of the symbol values, if you select the Vsym (Value-ordered symbol table) option. The Assembler requires an extra two bytes per symbol, over the size of the symbol table at the end of pass two, to complete this successfully. The default value is NOVsym.

When available memory is not sufficient, the Assembler uses what is available and prints only the symbols that it can sort. If you have a large assembly that uses up most of the 27K of symbol table space, the value-ordered symbol table may not contain all the identifiers.

Sixup Option The Sixup (Sixup symbol dump) option causes the Assembler to print the symbol table dump in six columns, instead of the normal four, when the listing is directed to a printer. Your printer must be able to print 12Ø or more characters per line. The four-column default results in a table that is 8Ø characters across.

The default value is NOSixup (option off).

Exp Option The Exp (Expansion lines) option causes printing of macro expansion lines. To suppress printing of macro expansion lines, use LST NOExp.

REP

REP expression

The REP (REPeat) directive is used to print a string of characters in your listings, starting at the first character of the source statement portion of the listing.

Unless you use the CHR directive (described below), the character printed is the asterisk (*). You can use any number of REP directives in a source file.

The REP directive is used to make listings more readable, typically by printing a string of asterisks to set off comment headings at the beginning of subroutines or modules. You can save considerable space in your source file by using this directive instead of simply inserting the string of asterisks in the file.

Only the low byte of the expression is used. You can have the currently defined CHR character printed up to 256 times. If you specify Ø or 256, 256 characters will be printed.

CHR

```
CHR "?"
```

The CHR (CHaRacter) directive is used to change the character repeated by the REP directive described above.

? represents any character you want printed instead of an asterisk. This directive can be used any number of times to change the character for different parts of your listing.

SKP

```
SKP n
```

The SKP (SKiP) directive lets you insert n blank lines in the listing, by sending ASCII carriage returns to the output device.

The device must provide its own line feed on CR if that device requires a LF to advance a print line on the paper.

SBTL

```
SBTL .string.
```

The SBTL (SuBTitLe) directive provides a title line (specified above as .string.) at the top of each page of the listing file.

The subtitle can be up to 20 characters long when output is to a 40-column screen, and up to 35 characters long when output is to an 80-column device. SBTL is optional, but it provides an easy way to identify a listing. This directive causes the first line of each subsequent page to contain the current subtitle, followed by the date and time. If no clock card is available to set the ProDOS time word to the current time, the Assembler prints blanks instead of the time.

In addition to setting a new subtitle, SBTL also causes a page break.

Using Macros in Assembly-Language Programs

The MACLIB directive enables you to use the Assembler's disk-based MACRO capability, and tells the Assembler on which disk the macro definitions are stored. The Assembler supports macro definitions that you have

created before the assembly and stored on disk. You cannot define macros within your program source file.

MACLIB pathname

where pathname is the pathname of the subdirectory that contains the MACRO files.

Each macro is a separate ProDOS text file. Before assembling, set the prefix to the path of the subdirectory that contains the macro files.

When you use the MACLIB directive in your program source file, you tell the Assembler that you may use macros later in your source file. Once macros are enabled, the Assembler assumes that any mnemonic you use in your source file that is not a standard 65Ø2 instruction or Assembly directive mnemonic is the name of a macro definition that resides on this disk.

Invoking Macros in a Source File

To invoke a macro in your program source file, include the name of the macro in the mnemonic field of an assembly statement, just as you would any other mnemonic. This macro name must be the name of a ProDOS text file that resides on the disk volume you specified in your earlier MACLIB directive. You can include operand expressions in the operand field of these assembly statements, but you cannot include a comment.

When the Assembler encounters a mnemonic that does not match either a standard 65Ø2 instruction or an Assembly directive, the Assembler suspends its assembly of the current source file, preserves any necessary information, and attempts to read a file from the macro disk having the same mnemonic name as the file name. If this macro definition file doesn't exist, the Assembler signals the error and cancels the assembly.

The Macro Definition File

The macro definition file is a text file consisting of regular 65Ø2 assembly statements. There are no special MACRO directives that you must use in defining a macro. You can specify string parameters that the Assembler will use in expanding the macro invocation in your source program.

When you invoke a macro in your program source file, you can specify up to nine string parameters in the operand field of the statement containing the macro name. These string parameters must be separated by commas. The Assembler parses this operand field, using the comma as the delimiter, and substitutes these string parameters into your macro definition where you have specified. Two commas with no characters between them signify a null parameter. You cannot pass a comma as part

of a string parameter except as a numeric constant; that is, as $2C or $AC.

In the macro definition file that you previously created on the macro source disk, you can indicate where these string parameters are to be substituted by using the two-character sequences &1 through &9. The Assembler replaces these two-character sequences with the characters of the first through ninth string parameters, respectively. If you reference a parameter in your macro definition that you did not supply when you invoked the macro in your source file, the Assembler replaces that parameter with zero (no) characters. You can use any number of '&n' parameters within a macro statement or within a single field of a statement.

Here is an example of a macro definition:

```
        LDA   &1
        CLC
        ADC   &2
        STA   &3
        LDA   &1+1
        ADC   &2+1
        STA   &3+1
```

If you save this sample macro definition into a file with the name ADD16, you can invoke the macro as shown in this example:

```
            MACLIB      ; ENABLE MACROS
  VALU1     EQU    $FA
  VALU2     EQU    $FB
  SUM       EQU    $FC
  ****************
 .
 .
  ; INVOKE ADD16 MACRO BELOW
            ADD16    VALU1,VALU2,SUM
 .
 .
```

When you assemble this source program, the Assembler will expand this macro and substitute the three parameters into the macro definition, where you have indicated. This macro expansion will appear in your assembly listings as shown in the following example:

```
 1          MACLIB          ; ENABLE MACROS
 2 VALU1    EQU    $FA
 3 VALU2    EQU    $FB
 4 SUM      EQU    $FC
 5 ****************
   .
   .
1Ø ; INVOKE ADD16 MACRO BELOW
11          ADD16  VALU1,VALU2,SUM
 1+         LDA    VALU1
 2+         CLC
 3+         ADC    VALU2
 4+         STA    SUM
 5+         LDA    VALU1+1
 6+         ADC    VALU2+1
 7+         STA    SUM+1
   .
   .
   .
```

The Assembler always lists the macro expansion following the assembly statement that invokes the macro (line 11). The lines of an expanded macro are indicated in the assembly listing by the plus character (+) following the line number. To suppress the printing of macro expansion lines, use the LST NOEXP directive.

Note that the Assembler substituted the &1 argument with the first string parameter (VALU1), and so on for all of the macro parameters. The process of substituting macro parameters must not produce an assembly statement longer than 255 characters. This can happen if you attempt to insert a large number of long string parameters into a long macro statement. You may also encounter this problem if you use a macro statement with a long comment field.

The Assembler supports two special features to help you use macros. These are the &Ø and &X parameters. The following paragraphs describe how you can use these parameters within your macro definitions.

The &Ø Parameter

You can use the &Ø parameter in your macro definition to represent the number of parameters present in the assembly statement that invokes this macro. The Assembler always counts the number of parameters present in the operand field of the statement invoking the macro, and substitutes this single-digit number wherever it finds the &Ø parameter in your macro definition.

You would typically use this &Ø parameter within a conditional assembly statement in your macro definition, either to validate the macro invocation or to create flexible macro definitions.

The &X Parameter

You can use the &X parameter (uppercase X only) in your macro definitions to represent a cumulative count of the number of times that you used any macro during this assembly. The Assembler substitutes a 1- to 4-digit numeric string for each occurrence of the &X parameter, starting with the string 1 and increasing this number by one each time you invoke a macro during your assembly.

The &X parameter lets you automatically generate unique labels within a macro expansion, even if you invoke the macro definition many times. All of the labels you generate using the &X parameter appear as individual entries in the symbol table and appear in the symbol table dump listing.

You can use the &X parameter in your macro definition by appending it to some label prefix or embedding it inside a label, as shown in this example:

```
1        LDY #5
2 F&XL   STA &1,Y
3        DEY
4        BNE F&XL
```

This example shows &X embedded in a label. The Assembler will create the unique labels F1L, F2L, and F29L when this macro is used as the first, second, and twenty-ninth macros of an assembly.

Chapter 4

The Bugbyter Debugger

Chapter 4

The Bugbyter Debugger

Chapter 4

The Bugbyter Debugger

About This Chapter

Every assembly-language programmer eventually runs up against a newly written or revised program that just doesn't work as intended. This isn't any reflection on the programmer; everyone occasionally overlooks something or omits an essential instruction, resulting in a program "bug."

This chapter describes how to use the Bugbyter program to test your assembly-language programs and eliminate errors. Testing and changing a program to eliminate errors is called **debugging.**

By the Way: According to computer pioneer Grace Murray Hopper, the first computer bug was discovered at Harvard in 1945. Hopper and her associates were having trouble with the Mark I, the first large-scale digital computer. The problem turned out to be a moth that had died in the circuits. Ever since, computer problems have been called bugs.

A real test of your programming skill is how quickly you can locate problems in your programs and fix errors to produce a working program. With the proper tools, testing and fixing a program should be easy and efficient, allowing you to produce error-free, quality software.

The Bugbyter program is a powerful display-oriented debugging tool that can save you considerable time in testing and debugging your assembly-language programs. Using Bugbyter for just a few minutes, you can observe precisely how your program is executing, and to locate where things are going wrong.

The Bugbyter program is also a useful tool for testing and verifying a working program to make sure that it will operate correctly under a variety of conditions.

This chapter consists of three main parts:

- An Overview that introduces the features of the Bugbyter program, describes how Bugbyter can help you debug your programs, and describes some restrictions on the use of Bugbyter.

- A series of tutorials on using Bugbyter, in which you test the assembly-language program you created in the tutorials in the previous chapters.

- A Reference Section describing in detail how to use all of Bugbyter's features while testing and debugging your programs.

In addition, Appendix C contains a summary of all of Bugbyter's functions and commands.

Overview

Bugbyter lets you load and control the execution of your assembly-language program on any Apple II system. You can use Bugbyter to test and debug almost any assembly-language program, as long as your Apple II's memory has enough room for both your program and Bugbyter. For the greatest flexibility in debugging assembly-language programs, you can load and execute the Bugbyter program almost anywhere in memory. This relocation feature is discussed later in this chapter.

Bugbyter allows you a variety of options to use when debugging your 65Ø2 assembly-language programs.

At any time while you are debugging your program, you can view the status of the 65Ø2 registers, stack, and memory as they appear to your assembly-language program.

When you are debugging internal portions of your program and you do not need to view the programmed displays, you can use Bugbyter's Master Display to show you

- All of the Apple's 65Ø2 internal registers

- A portion of the Apple II's program stack

- A mnemonic disassembly of portions of your assembly-language program

- Portions of your computer's memory
- Any real or transparent breakpoints that you may set.

Using Bugbyter, you can step through your program one instruction at a time, observing all the effects of executing each instruction. If you want, you can alter the layout of Bugbyter's Master Display to show you more or fewer stack or memory locations, program statements, or breakpoints.

At any time, you can also change the conditions under which you are testing or debugging your program. For example, you can

- Change the contents of any 65Ø2 register
- Alter the contents of memory locations or the stack to change data values that may be stored there
- Alter instructions or parts of your program and immediately test any revisions in your program.

To quickly fix or change the 65Ø2 assembly-language instructions that make up your program, you can enter 65Ø2 assembly-language mnemonics directly into your Apple II's memory. Bugbyter translates these mnemonic instructions and stores the actual machine codes into the memory locations that you select.

To test programs that use real-time operating system routines, or to test portions of your own programs that must execute in real-time, you can indicate to Bugbyter regions of code that must execute in real-time. Bugbyter allows you to test these programs using all of its debugging facilities, while still allowing these portions of your program to execute at the full speed of the 65Ø2 CPU.

Using Bugbyter's Master Display and the other features of this debugging tool, you will be able to quickly test, debug, and fix your assembly-language programs.

Restrictions on Using Bugbyter

You can use Bugbyter to test and debug any assembly-language program as long as your own program leaves a contiguous block of memory 694Ø ($1B1C) bytes long somewhere in your Apple II's memory. This memory is needed to contain the Bugbyter program code and data areas. Bugbyter also uses the first 32 ($2Ø) bytes of the Apple II's stack (memory locations $1ØØ to $11F), although this should not cause problems unless your own program alters the contents of the beginning of the stack.

WARNING
Unless you specify a starting address with the BRUN command, the Bugbyter program will load starting at memory address $2000.

The section "Relocating the Bugbyter Program" (later in this chapter) explains how to relocate Bugbyter to prevent any conflicts with your own program.

If programs for your Apple IIe system use bank-switching to extend the usable memory, Bugbyter must remain resident in memory at all times. Although you can debug programs that use bank-switching, you cannot swap Bugbyter out of the usable memory space while you are debugging.

Allowing for these memory restrictions, there are few programs that Bugbyter cannot be used to test and debug.

Tutorials

These short tutorials show you how to use Bugbyter and many of its commands to test the operation of the short program you created and assembled in the tutorials in the previous chapters. You will test your program to verify that it works as you intended, and modify it if necessary to make it execute differently.

To use these tutorials, you need

- An Apple II system.
- The ProDOS Assembler Tools disk containing the Bugbyter program and the TESTPROGRAM.0 binary (BIN) file that you created as part of the tutorial in Chapter 3.

These tutorials teach you how to

- Start Bugbyter
- Use Bugbyter's Master display
- Load your assembly-language object program from disk
- Test your program using the Single-Step and Trace modes
- Change your program and test it again.

Tutorial 1. Getting Started

1. Insert the ProDOS Assembler Tools system disk into disk drive 1, and turn on your Apple II. After the ProDOS operating system is loaded from the disk, the Applesoft prompt (]) appears.

2. Type

```
BRUN BUGBYTER
```

   and press RETURN. Your Apple II loads the Bugbyter program.

The Master Display

Once Bugbyter is loaded, its Master Display appears on the screen. This display is divided into six subdisplays, all described on the following pages.

With Bugbyter loaded, you are in Bugbyter Command Level. You should see a display similar to the one shown on the next page:

```
C    R  B  PC   A  X  Y  S  P  NV-BDIZC
ØØØØ ØØ O  ØØØØ ØØ ØØ ØØ FF Ø2 ØØØØØØ1Ø

 1F9: C1
 1FA: F1
 1FB: ØØ
 1FC: Ø1
 1FD: Ø1
 1FE: 9D
 1FF: 37
 1ØØ: FF
 1Ø1: FF
 1Ø2: FF
 1Ø3: FF
 1Ø4: FF
 1Ø5: FF

ØØØØ:4C L  BP  POINT COUNT TRIG  BROKE
ØØØØ:4C L  1   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  2   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  3   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  4   ØØØØ  ØØØØ  ØØØØ  ØØØØ

:(C) 1982 COMPUTER-ADVANCED IDEAS  V2.Ø3
```

Your Bugbyter Master Display may be slightly different from the one illustrated.

The Register Subdisplay

In the Register subdisplay at the top of the screen, Bugbyter displays the six 65Ø2 registers and three Bugbyter registers.

```
C    R  B  PC   A  X  Y  S  P  NV-BDIZC
ØØØØ ØØ O  ØØØØ ØØ ØØ ØØ FF Ø2 ØØØØØØ1Ø
```

The 65Ø2 registers are

- PC (Program Counter)
- A (A-Register)
- X (X-Register)
- Y (Y-Register)
- S (Stack pointer)
- P (Processor Status Register)

In the upper right corner of this display, Bugbyter displays the Processor Status Register in both two-digit Hexadecimal notation (P) and binary (NV-BDIZC), where the individual flags are

- N (the Negative flag)
- V (the Overflow flag)
- B (the Break flag)
- D (the Decimal flag)
- I (the Interrupt flag)
- Z (the Zero flag)
- C (the Carry flag)

The Bugbyter registers in the upper left corner of the display are explained later in this Chapter. They are

- C (the Cycle Count)
- R (the Trace Rate)
- B (the Breakpoint flag, either O (Out) or I (In))

The Stack Subdisplay

Bugbyter's Stack subdisplay is a window into the 65Ø2 memory stack. This Stack subdisplay contains the ascending addresses of memory locations just before and after the location pointed to by the 65Ø2 Stack Pointer, showing the contents of each byte in a portion of the stack:

```
1F9: C1
1FA: F1
1FB: ØØ
1FC: Ø1
1FD: Ø1
1FE: 9D
1FF: 37
1ØØ: FF
1Ø1: FF
1Ø2: FF
1Ø3: FF
1Ø4: FF
1Ø5: FF
```

Bugbyter always highlights the line in this subdisplay that represents the current location of the 65Ø2 Stack Pointer.

The Code Disassembly Subdisplay

To the right of the Stack subdisplay is the Code Disassembly subdisplay. Bugbyter uses this window to display a disassembly of your program's code, using standard 65Ø2 assembly mnemonics and address mode syntax.

The Disassembly subdisplay now on your screen is blank. The first line of the Disassembly subdisplay might read:

```
1ØØØ: LDY #$CØ     AØ CØ
```

where

1ØØØ	is the address of the instruction
LDY	is the 65Ø2 instruction mnemonic
#$CØ	represents the instruction operand
AØ CØ	are the actual machine-instruction bytes in memory.

The actual machine-instruction bytes (AØ CØ) are displayed as one of Bugbyter's information display options. As you trace or single-step through your program, you can select one of seven display options for Bugbyter to display. These options are described later in this chapter, in the section on Trace mode.

The Memory Cell Subdisplay

In its Memory Cell subdisplay, Bugbyter displays the contents of a number of memory locations that may be important to your program. You can use the MEM command (described later, in the sections on viewing and altering memory) to select the addresses of individual bytes or

byte-pairs that Bugbyter will display continuously in this portion of the Master Display.

```
Ø000:4C L
Ø000:4C L
Ø000:4C L
Ø000:4C L
Ø000:4C L
```

The Breakpoint Subdisplay

Bugbyter allows you to set a number of program breakpoints that you can use to control the execution of your program. Bugbyter displays these breakpoints and relevant breakpoint information in the Master Display's Breakpoint subdisplay, just below the Code Disassembly subdisplay.

```
BP  POINT COUNT TRIG  BROKE
1   Ø000  Ø000  Ø000  Ø000
2   Ø000  Ø000  Ø000  Ø000
3   Ø000  Ø000  Ø000  Ø000
4   Ø000  Ø000  Ø000  Ø000
```

Breakpoints and other debugging techniques are described in the section "Controlling the Execution of Your Program," later in this chapter.

The Bugbyter Command Line

On the last line of the display is the Bugbyter command line. It now contains a brief copyright notice. The blinking cursor just after the colon (:) prompt signifies that Bugbyter is ready to accept commands from you.

Having loaded the Bugbyter program, you are in the Bugbyter command level. You can reach many of Bugbyter's features directly from command level. To reach others, you must use commands that place you in one of Bugbyter's three debugging modes.

Outlined on the next page are the five Bugbyter modes and the things that you can do in each mode.

These are the five Bugbyter modes:

1. Bugbyter Command Level, in which you can do the following:

   - Execute ProDOS commands, for example BLOAD programs.

   - View or alter contents of memory, entering hexadecimal values, character codes, or 65Ø2 assembly code.

   - Change the contents of the 65Ø2 or Bugbyter registers.

   - Enter Applesoft/ProDOS or the Monitor.

   - Alter the layout of the Bugbyter Master Display.

   - Set and clear breakpoints.

   - View areas of memory, using Bugbyter's Memory Page Display.

   - Enter one of Bugbyter's three debugging modes.

2. Memory Page Display Mode, in which you can:

   - View or alter contents of memory, entering hexadecimal values, character codes, or 65Ø2 assembly code.

3. Single-Step Mode, in which you can:

   - Single-step, or execute your program one instruction at a time.

   - View Bugbyter's Master Display, or your Apple II's low-resolution, high-resolution, or normal text screens.

4. Bugbyter Trace Mode, in which you can:

   - Trace your program's execution, updating Bugbyter's Master Display after each instruction, scanning for RTS opcodes or transparent breakpoints.

   - View Bugbyter's Master Display, or your Apple II's low-resolution, high-resolution, or normal text screens.

5. Bugbyter Execution Mode, in which you can:

   - Execute your program directly on the 65Ø2, using real breakpoints to control execution.

Typing and Editing Bugbyter Commands

To enter a command at the Bugbyter Command Level, simply type the characters to form the command, then press RETURN.

To make a correction, use the LEFT ARROW or RIGHT ARROW key to place the cursor over the incorrect character. Then you can

- delete the character, by pressing CONTROL-D
- replace the character with a different one, by typing a new character over the old one.

To move the cursor to the beginning of the command line, press CONTROL-B; to move it to the end of the line, press CONTROL-N.

To insert characters, use the LEFT ARROW or RIGHT ARROW key to move the cursor to the character before which the insertion is to be made. Press CONTROL-I, and type the new characters. End the insertion by pressing LEFT ARROW, RIGHT ARROW, or RETURN.

To enter a control character onto the command line, for storing character-string information into memory, press CONTROL-C before typing the control character.

To erase an erroneous line, press CONTROL-X.

When you have finished editing the line, press RETURN. This causes Bugbyter to accept all of the characters on the command line.

Before You Press RETURN: If there are unwanted characters to the right of the cursor, use CONTROL-D to delete them, or press the SPACE bar to replace them with spaces.

The following table summarizes Bugbyter's editing functions and the keystrokes that invoke them.

Editing Function	Keystroke
Move cursor left one character	LEFT ARROW
Move cursor right one character	RIGHT ARROW
Move cursor to beginning of line	CONTROL-B
Move cursor to end of line	CONTROL-N
Delete entire line	CONTROL-X
Delete current character	CONTROL-D
Insert characters	CONTROL-I
Enter a control character	CONTROL-C, then any character
Accept the current line as typed	RETURN

Tutorial 2. Loading Your Program

Having loaded Bugbyter and viewed the Master Display, you need to load your program into memory. You will use the ProDOS BLOAD command to do this. You will then be able to use Bugbyter to debug your program. To enter any ProDOS command while you are using Bugbyter, simply type a period (.) before you type the command.

1. Type

   .BLOAD TESTPROGRAM.Ø

   Remember to type a period (.) before typing the ProDOS command. TESTPROGRAM.Ø is the binary file you created when you did the tutorial in Chapter 2.

   After your Apple II has loaded TESTPROGRAM.Ø into memory, you will again see the blinking cursor.

2. To view the program that you just loaded into memory, type

   1ØØØL

   and press RETURN.

   If you type 1ØØØL, Bugbyter fills the Disassembly portion of your screen with the first few instructions of your program, starting with the instruction at address $1ØØØ.

   As TESTPROGRAM.Ø is a short program, you will be able to see the entire program on the screen at once. It should look similar to the display on the next page.

```
1000: LDY #$C0       A0 C0
1002: LDX #$00       A2 00
1004: JSR $100D      20 0D 10
1007: INX            E8
1008: CPX #$05       E0 05
100A: BNE $1004      D0 F8
100C: RTS            60
100D: INY            C8
100E: TYA            98
100F: STA $1100,X    9D 01 11
1012: RTS            60
1013: BRK            00
1014: BRK            00
```

3. Check the instructions in your program to verify that your program was assembled correctly in the previous chapter. Don't worry if you see different values in the Stack subdisplay; this is normal.

Tutorial 3. Single-Stepping Through Your Program

Bugbyter lets you watch the execution of your program one instruction at a time, seeing the results of each instruction.

1. To begin single-stepping your program at address $1000, type

   1000S

   Bugbyter highlights the LDY instruction that appears at address $1000, and the program counter (PC) on the top line of the screen shows the address 1000.

```
-------------------------------------------------------------

 C    R  B  PC   A  X  Y  S  P  NV-BDIZC
 ØØØØ ØØ O  1ØØØ ØØ ØØ ØØ FF Ø2 ØØØØØØ1Ø

  1F9: C1
  1FA: F1
  1FB: ØØ
  1FC: Ø1
  1FD: Ø1
  1FE: 9D
  1FF: 37
  1ØØ: FF
  1Ø1: FF
  1Ø2: FF
  1Ø3: FF   1ØØØ: LDY #$CØ
  1Ø4: FF   1ØØ2: LDX #$ØØ
  1Ø5: FF   1ØØ4: JSR $1ØØD

-------------------------------------------------------------
```

Note that no instructions of your program have been executed, and none of the registers has been changed, except the PC register. The first instruction to be executed is to load the Y-register with the value $CØ, as indicated by the highlighted LDY instruction on the Master Display.

2. To execute this LDY instruction, press the SPACE bar. Bugbyter executes this instruction and updates the display.

```
C    R  B  PC   A  X  Y  S  P  NV-BDIZC
ØØØØ ØØ 0  1ØØ2 ØØ ØØ CØ FF BØ 1Ø11ØØØØ

 1F9: C1
 1FA: F1
 1FB: ØØ
 1FC: Ø1
 1FD: Ø1
 1FE: 9D
 1FF: 37
 1ØØ: FF
 1Ø1: FF
 1Ø2: FF   1ØØØ: LDY #$CØ     P:1Ø11ØØØ
 1Ø3: FF   1ØØ2: LDX #$ØØ
 1Ø4: FF   1ØØ4: JSR $1ØØD
 1Ø5: FF   1ØØ7: INX

ØØØØ:4C L  BP  POINT COUNT TRIG  BROKE
ØØØØ:4C L  1   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  2   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  3   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  4   ØØØØ  ØØØØ  ØØØØ  ØØØØ

 SINGLE STEP
```

Note that the program counter has increased to 1ØØ2, and that register Y has been loaded with the hex value CØ. In the Disassembly subdisplay, the disassembled instructions have shifted and the highlighted bar is now highlighting the instruction at address 1ØØ2, the next instruction to be executed. Bugbyter has also displayed the processor status register (in binary) to the right of the instruction that was just executed.

3. The next highlighted instruction loads the X-register with the value $ØØ. Press the SPACE bar to execute this instruction.

```
C    R  B  PC   A  X  Y  S  P  NV-BDIZC
ØØØØ ØØ 0  1ØØ4 ØØ ØØ CØ FF 32 ØØ11ØØ1Ø

 1F9: C1
 1FA: F1
 1FB: ØØ
 1FC: Ø1
 1FD: Ø1
 1FE: 9D
 1FF: 37
 1ØØ: FF
 1Ø1: FF   1ØØØ: LDY #$CØ     P:1Ø11ØØØØ
 1Ø2: FF   1ØØ2: LDX #$ØØ     P:ØØ11ØØ1Ø
 1Ø3: FF   1ØØ4: JSR $1ØØD
 1Ø4: FF   1ØØ7: INX
 1Ø5: FF   1ØØ8: CPX #$Ø5

ØØØØ:4C L  BP  POINT COUNT TRIG  BROKE
ØØØØ:4C L  1   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  2   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  3   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  4   ØØØØ  ØØØØ  ØØØØ  ØØØØ

SINGLE STEP
```

4. Verify that the X-register now contains the value $ØØ. The next instruction is a jump-to-subroutine instruction that calls the STORE subroutine at address 1ØØD. This JSR instruction will change the contents of the stack pointer as well as the program counter. Press the SPACE bar to execute this instruction.

   Bugbyter executes this instruction and calls the subroutine at 1ØØD. Note that the program counter (PC) contains the address 1ØØD.

   Note that the stack pointer (S) has changed to FD and that the stack subdisplay has shifted down. The stack pointer points to the location identified by the highlighted bar. Also note that the two-byte address that has been pushed onto the stack (the Ø6 and 1Ø bytes just below the highlighted bar) form the low-byte and high-byte of the address (1ØØ6) of the last byte of the JSR instruction.

```
C    R  B  PC   A  X  Y  S  P  NV-BDIZC
ØØØØ ØØ 0  1ØØ4 ØØ ØØ CØ FF 32 ØØ11ØØ1Ø

 1F7: 43
 1F8: D4
 1F9: C1
 1FA: F1
 1FB: ØØ
 1FC: Ø1
 1FD: Ø1
 1FE: Ø6    1ØØØ: LDY #$CØ     P:1Ø11ØØØØ
 1FF: 1Ø    1ØØ2: LDX #$ØØ     P:ØØ11ØØ1Ø
 1ØØ: FF    1ØØ4: JSR $1ØØD    P:ØØ11ØØ1Ø
 1Ø1: FF    1ØØD: INY
 1Ø2: FF    1ØØE: TYA
 1Ø3: FF    1ØØF: STA $11ØØ,X

ØØØØ:4C L  BP  POINT COUNT TRIG  BROKE
ØØØØ:4C L  1   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  2   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  3   ØØØØ  ØØØØ  ØØØØ  ØØØØ
ØØØØ:4C L  4   ØØØØ  ØØØØ  ØØØØ  ØØØØ

SINGLE STEP
```

5. Execute the next two instructions by pressing the SPACE bar twice (once for each instruction). These instructions increment the contents of the Y-register and transfer the contents of the Y-register to the A-register. To verify the operation of these instructions, watch the Register subdisplay.

Tutorial 4. Using the Memory Subdisplay

The next instruction to be executed is STA $11ØØ,X, which stores the value in the A-register into memory address $11ØØ (if you were to press the SPACE bar right now, you would execute this instruction and store the value $C1 into memory). Before executing this instruction, though, we will use the Memory subdisplay to verify that the proper value gets stored in memory.

1. Press ESC to exit from Bugbyter's Single-Step mode and return to Command Level. You should see the colon command prompt and blinking cursor of Bugbyter's Command Level.

2. Type MEM and press RETURN. The blinking cursor appears in the first position of the Memory subdisplay.

   Each cell of the Memory subdisplay consists of an address, followed by a colon and the hex value of the byte stored in that memory location. After the hex value is the Apple character that corresponds to that hex value.

```
ØØØØ:4C L
ØØØØ:4C L
ØØØØ:4C L
ØØØØ:4C L
ØØØØ:4C L

:MEM
```

3. To set a particular memory address in this first memory cell, type 11ØØ. The address 11ØØ appears in the first memory cell of the Memory subdisplay, followed by the byte currently stored in that memory location. The blinking cursor moves to the next memory cell.

4. To enter the addresses of the next four memory locations, type 11Ø1 11Ø2 11Ø3 11Ø4. You now have loaded the addresses of five consecutive memory locations into the Memory subdisplay. (As you later execute your test program, your program will fill each of these memory locations with an Apple character code.)

5. If you make a mistake typing the addresses, don't worry--just press RETURN to move the blinking cursor to the memory cell with the address you want to change, and type the new address over the old. When you are finished, press ESC to return to the Bugbyter Command Level.

6. To return from the Bugbyter Command Level to Single-Step mode and execute the next instruction, type S and press RETURN. Bugbyter executes the next instruction of your program, storing the value $C1 into the memory location $11ØØ. To verify that this value actually was stored, look at the value shown in the Memory subdisplay for that address.

```
11ØØ:C1 A
11Ø1:ØØ @
11Ø2:ØØ @
11Ø3:ØØ @
11Ø4:ØØ @

SINGLE STEP
```

Tutorial 5. Tracing Your Program

To make sure that the rest of your program operates as it should, we will trace the rest of the program, rather than single-step through it. When Bugbyter traces your program, Bugbyter updates the Master Display after executing each instruction. You can watch your program fill the memory location $11Ø1 to $11Ø4 with the codes for the characters "B" through "E."

1. To trace the remainder of your program, press RETURN while you are in Bugbyter Single-Step mode. You should see the words SINGLE STEP on the Bugbyter command line before pressing RETURN. After you press RETURN, Bugbyter replaces this with the word TRACE (for Trace mode).

   Bugbyter TRACEs the rest of your program, executing each instruction in turn. You can watch the Bugbyter Master Display change as Bugbyter executes your program. When Bugbyter finishes executing your program, it will stop.

2. To verify that your program correctly loaded the five characters "A" through "E" into the memory locations $11ØØ through $11Ø4, look at the Memory subdisplay.

```
11ØØ:C1 A
11Ø1:C2 B
11Ø2:C3 C
11Ø3:C4 D
11Ø4:C5 E

:_
```

Tutorial 6. Changing Your Program in Memory

You can use Bugbyter to change your program in memory. This lets you immediately test new versions of your program without having to leave Bugbyter and use the Editor/Assembler to edit and reassemble your source program.

For example, let's change the first instruction of your program, at address 1ØØØ, to LDY #$ØØ.

1. At the Bugbyter Command Level, type

   1ØØØ: LDY #$ØØ

   and press RETURN.

   Note that you must follow the address you have typed with a colon (:) before you type the assembly-language instruction mnemonic.

2. To see how your entire program looks now, type

   1ØØØL

   and press RETURN.

   Notice that address $1ØØØ now contains the new instruction.

   It is also possible to change just one byte in your program. Notice from the disassembly subdisplay that address $1ØØ9 (the count limit used in controlling the program loop) has the value $Ø5. To change the number of times that your program runs through this loop, let's change this limit to $CØ.

3. At the Bugbyter Command Level, type

   1ØØ9: CØ

   and press RETURN.

   Note again that you must use a colon after the address as you did when changing an entire instruction.

4. To view your new program, type

   1ØØØL

   and press RETURN.

   Bugbyter displays your new program in its Disassembly subdisplay:

```
C    R  B  PC   A  X  Y  S  P  NV-BDIZC
ØØØØ ØØ 0  1ØØC C5 Ø5 C5 FF 33 ØØ11ØØ11

 1F9: C1   1ØØØ: LDY #$ØØ      AØ ØØ
 1FA: F1   1ØØ2: LDX #$ØØ      A2 ØØ
 1FB: ØØ   1ØØ4: JSR $1ØØD     2Ø ØD 1Ø
 1FC: Ø1   1ØØ7: INX            E8
 1FD: Ø1   1ØØ8: CPX #$CØ       EØ CØ
 1FE: 9D   1ØØA: BNE $1ØØ4      DØ F8
 1FF: 37   1ØØC: RTS            ØØ
 1ØØ: FF   1ØØD: INY            C8
 1Ø1: FF   1ØØE: TYA            98
 1Ø2: FF   1ØØF: STA $11ØØ,X    9D Ø1 11
 1Ø3: FF   1Ø12: RTS            6Ø
 1Ø4: FF   1Ø13: BRK            ØØ
 1Ø5: FF   1Ø14: BRK            ØØ

11ØØ:C1 A  BP  POINT COUNT TRIG  BROKE
11Ø1:C2 B  1   ØØØØ  ØØØØ  ØØØØ  ØØØØ
11Ø2:C3 C  2   ØØØØ  ØØØØ  ØØØØ  ØØØØ
11Ø3:C4 D  3   ØØØØ  ØØØØ  ØØØØ  ØØØØ
11Ø4:C5 E  4   ØØØØ  ØØØØ  ØØØØ  ØØØØ

:_
```

5. To trace this new version of your program, type

   1ØØØT

   and press RETURN.

   Bugbyter begins tracing your new program, starting at location $1ØØØ.

It now takes Bugbyter about 3Ø seconds to trace your entire program, as you have made your program execute the program loop many more times than before. You can watch the X-register count how many times your program has gone through the loop.

Tutorial 7. Viewing a Page of Memory

The modified version of your program fills nearly a page of memory with the first 192 bytes of the Apple character set. You can use Bugbyter's Memory Page Display to view this area of memory.

1. When Bugbyter has finished tracing your program, type

   11ØØ:

   and press RETURN.

   Bugbyter replaces the Master Display with its Memory Page Display: a screen of memory information for memory starting at location $11ØØ. Because your program executed the loop $CØ (decimal 192) times, you will see much of the Apple character set displayed, including inverse-video and blinking characters.

```
11ØØ:  Ø1 Ø2 Ø3 Ø4 Ø5 Ø6 Ø7 Ø8  ABCDEFGH
11Ø8:  Ø9 ØA ØB ØC ØD ØE ØF 1Ø  IJKLMNOP
111Ø:  11 12 13 14 15 16 17 18  QRSTUVWX
1118:  19 1A 1B 1C 1D 1E 1F 2Ø  YZ[\]^
112Ø:  21 22 23 24 25 26 27 28  !"#$%& (
1128:  29 2A 2B 2C 2D 2E 2F 3Ø  )*+,-./Ø
113Ø:  31 32 33 34 35 36 37 38  12345678
1138:  39 3A 3B 3C 3D 3E 3F 4Ø  9:;<=>?@
114Ø:  41 42 43 44 45 46 47 48  ABCDEFGH
1148:  49 4A 4B 4C 4D 4E 4F 5Ø  IJKLMNOP
115Ø:  51 52 53 54 55 56 57 58  QRSTUVWX
1158:  59 5A 5B 5C 5D 5E 5F 6Ø  YZ[\]^
116Ø:  61 62 63 64 65 66 67 68  !"#$%& (
1168:  69 6A 6B 6C 6D 6E 6F 7Ø  )*+,-./Ø
117Ø:  71 72 73 74 75 76 77 78  12345678
1178:  79 7A 7B 7C 7D 7E 7F 8Ø  9:;<=>?@
118Ø:  81 82 83 84 85 86 87 88  ABCDEFGH
1188:  89 8A 8B 8C 8D 8E 8F 9Ø  IJKLMNOP
119Ø:  91 92 93 94 95 96 97 98  QRSTUVWX
1198:  99 9A 9B 9C 9D 9E 9F AØ  YZ[\]^
11AØ:  A1 A2 A3 A4 A5 A6 A7 A8  !"#$%& (
11A8:  A9 AA AB AC AD AE AF BØ  )*+,-./Ø
11BØ:  B1 B2 B3 B4 B5 B6 B7 B8  12345678
:
```

2. To return to the Bugbyter Command Level, press ESC;

   or,

   to exit Bugbyter and return to Applesoft/ProDOS, type QUIT and press RETURN.

Using Bugbyter

The remainder of this chapter describes in detail each of Bugbyter's debugging functions (the preceding tutorials have already introduced you to some of these). They include

- Starting, relocating, and restarting Bugbyter
- Viewing and altering memory or 65Ø2 registers
- Altering Bugbyter's Master Display layout
- Controlling the execution of your program with debugging techniques such as

  Single-Step mode;
  Trace mode and transparent breakpoints
  Execution mode and real breakpoints

- Debugging real-time code
- Debugging programs that use the keyboard and display.

Bugbyter is a 6.8K ($1B1C bytes) binary program that must be fully resident in memory to be used to test or debug programs. Because your own program must share the Apple II's memory with Bugbyter, be careful that the two programs do not overlap each other in memory. Appendix H contains a memory map showing all the locations in which your assembly-language program and Bugbyter may reside.

Typically, you will simply use the ProDOS BRUN command to load and execute Bugbyter, just as you did in the tutorial. To recapitulate: from Applesoft, type

```
BRUN BUGBYTER
```

and press RETURN. Your Apple II reads the Bugbyter program from your disk and loads it into memory locations $2ØØØ to $3B1F.

Relocating the Bugbyter Program

If your own program uses memory locations within the block $2000 to $3B1F, you must relocate Bugbyter to some other memory location. You can specify any starting address from $800 to $7A00 when you type the BRUN command to execute Bugbyter. For example, to run Bugbyter starting at location $1000, type

```
BRUN BUGBYTER,A$1000
```

and press RETURN. Your Apple II loads Bugbyter and executes it from locations $1000 through $2B1F.

WARNING
Unless you specify a starting address with the BRUN command, the Bugbyter program will load starting at memory address $2000.

By the Way: The Bugbyter program is self-modifying; that is, once you BRUN the program at a certain address, Bugbyter modifies itself to allow itself to execute at that address. For this reason, if you want to move Bugbyter around in memory you should BLOAD Bugbyter before moving it, rather than BRUNning it.

For example, if the program you are debugging does not require ProDOS, you might want to run Bugbyter at address $A400 to leave more room for your own program. To do this, BLOAD Bugbyter at some lower address ($2000, for example), and then use the Monitor to move Bugbyter to start at address $A400.

Entering the Monitor

From the Bugbyter Command Level, you can enter the Apple Monitor, to use the Monitor's block memory moves and other capabilities not provided by Bugbyter. Your Apple II reference manual describes the features of the Apple II Monitor.

To enter the Apple Monitor's command mode, type

M

and press RETURN. After you have finished using the Monitor, return to the Bugbyter Command Level by pressing CONTROL-Y and RETURN.

Restarting Bugbyter

After exiting Bugbyter for any reason, either into the Monitor or into BASIC, you can use the Monitor's CONTROL-Y vector to restart Bugbyter, or you can use Bugbyter's load address.

For example, if you ran Bugbyter by typing

BRUN BUGBYTER,A$4ØØØ

and pressing RETURN, you can restart Bugbyter from Applesoft BASIC by typing either

CALL 1Ø16 (This uses the Monitor's CONTROL-Y vector.)

or

CALL 16384 (This uses Bugbyter's specific load address.)

and pressing RETURN.

From the Monitor, you can reenter Bugbyter by pressing CONTROL-Y and then RETURN.

Memory and the Bugbyter Displays

In this chapter's tutorials, you were introduced to some of Bugbyter's features for viewing and altering the contents of memory locations and registers. This section describes these functions in more detail, and describes how to alter the layout of Bugbyter's Master Display.

You can use Bugbyter in several different ways to view the contents of memory:

- The Stack and Disassembly subdisplays of Bugbyter's Master Display show the contents of specific regions of memory. To select the region that is shown in the Disassembly subdisplay, use the L (Load) command described in the tutorial.

- To use the Memory subdisplay of Bugbyter's Master Display to show the contents of several individual memory locations, use the MEM command (described below) to set particular memory addresses for each cell of this subdisplay.

- To display the contents of 184 ($B8) contiguous memory locations both as hexadecimal values and as Apple characters, use Bugbyter's Memory Page Display, described below.

Using the Memory Subdisplay

The Memory Cell subdisplay of Bugbyter's Master Display continuously displays the contents of several individual memory locations that you select. You can use the MEM command to set the memory addresses in one or more cells of this display. If you want, you can first use the SET command (described later in this section) to increase the number of cells in this subdisplay.

To set addresses in the Memory subdisplay from the Bugbyter command level, type

MEM

and press RETURN. Bugbyter moves the cursor to the first address at the top of the Memory subdisplay. You now have two options:

- Type a four-digit hexadecimal address for this memory cell, and press RETURN.

- Use the LEFT ARROW, RIGHT ARROW, or RETURN key to move the blinking cursor to another memory cell.

Unless you specify otherwise, Bugbyter displays the contents of each memory cell both as a hexadecimal value and as an Apple character.

To have Bugbyter instead display the contents of two consecutive locations (address+1 and address) as a four-digit hexadecimal address pointer (most significant byte first), type P before typing the address into a memory cell.

To return to the Bugbyter Command Level, press ESC.

Viewing the Memory Page Display

To view an entire screen of memory locations (two thirds of a 65Ø2 memory page), go to the Bugbyter Command Level and type the starting address of the memory block. Type a colon (:) and press RETURN.

For example, if you had just completed the tutorial and had the Apple character set in memory at address $11ØØ, typing 11ØØ: and pressing RETURN would command Bugbyter to replace the Master Display with the Memory Page Display having the address 11ØØ in the upper left corner.

```
1100:  01 02 03 04 05 06 07 08  ABCDEFGH
1108:  09 0A 0B 0C 0D 0E 0F 10  IJKLMNOP
1110:  11 12 13 14 15 16 17 18  QRSTUVWX
1118:  19 1A 1B 1C 1D 1E 1F 20  YZ[\]^
1120:  21 22 23 24 25 26 27 28  !"#$%& (
1128:  29 2A 2B 2C 2D 2E 2F 30  )*+,-./0
1130:  31 32 33 34 35 36 37 38  12345678
1138:  39 3A 3B 3C 3D 3E 3F 40  9:;<=>?@
1140:  41 42 43 44 45 46 47 48  ABCDEFGH
1148:  49 4A 4B 4C 4D 4E 4F 50  IJKLMNOP
1150:  51 52 53 54 55 56 57 58  QRSTUVWX
1158:  59 5A 5B 5C 5D 5E 5F 60  YZ[\]^
1160:  61 62 63 64 65 66 67 68  !"#$%& (
1168:  69 6A 6B 6C 6D 6E 6F 70  )*+,-./0
1170:  71 72 73 74 75 76 77 78  12345678
1178:  79 7A 7B 7C 7D 7E 7F 80  9:;<=>?@
1180:  81 82 83 84 85 86 87 88  ABCDEFGH
1188:  89 8A 8B 8C 8D 8E 8F 90  IJKLMNOP
1190:  91 92 93 94 95 96 97 98  QRSTUVWX
1198:  99 9A 9B 9C 9D 9E 9F A0  YZ[\]^
11A0:  A1 A2 A3 A4 A5 A6 A7 A8  !"#$%& (
11A8:  A9 AA AB AC AD AE AF B0  )*+,-./0
11B0:  B1 B2 B3 B4 B5 B6 B7 B8  12345678
:_
```

Your display might show different values, depending on the current contents of this region of memory.

Bugbyter displays the contents of memory in a table with eight memory locations to a line. The contents of each memory location are shown first in hexadecimal and then in their equivalent Apple characters. Each line of this table starts with the memory address of the first byte shown on that line. The normal Apple character set is:

00-3F	inverse-video characters
40-7F	flashing-video characters
80-FF	normal-video characters (two sets of alphabetic characters)

Bugbyter displays the colon prompt on the last line of this display and accepts either another address followed by a colon, to display another memory page, or a memory assignment command. The memory assignment command is described in the next section.

To return to the Bugbyter Command Level and the Master Display, press ESC.

Altering the Contents of Memory

You can use Bugbyter to change the contents of any RAM memory location or sequence of locations in your Apple II. From the Bugbyter Command Level or from the Memory Page Display, type a hexadecimal address and a colon, then one of the following:

- One or more hexadecimal bytes
- One or more character strings (enclosed in double or single quotation marks)
- Combinations of the two above
- 65Ø2 instruction mnemonics and operands.

You can freely mix hexadecimal numbers and character strings of any length in your memory assignment commands, separating the items with one or more spaces.

When you enter character strings, Bugbyter stores each character with its most significant bit on if you enclose the characters in double quotation marks ("). Bugbyter stores each character with its most significant bit off if you enclose the character string in single quotation marks (').

For example, if you type

```
8Ø5: "HELLO" 8D
```

and press RETURN, Bugbyter fills the memory locations from $8Ø5 to $8ØA with the bytes $C8, $C5, $CC, $CC, $C5 ("HELLO"), followed by a byte with the value of $8D. Because you used double quotation marks to delimit the character string, the character codes are stored with their most significant bit on.

Bugbyter also accepts 65Ø2 instruction mnemonics in a memory assignment command. You must use the standard address mode syntax when specifying operands, and you must type each assembly-language instruction in a separate memory assignment command. For example, if you type

```
1ØØØ: LDY #$8Ø
```

and press RETURN, Bugbyter assembles this statement and stores the two resulting bytes ($AØ and $8Ø) in the memory locations $1ØØØ and $1ØØ1.

Whenever you alter a memory location that is currently displayed on either the Bugbyter Master Display or the Memory Page Display, Bugbyter immediately updates the display to show the changes.

Altering the Contents of Registers

The Bugbyter Master Display always displays the current contents of the 65Ø2 microprocessor registers as they would appear to the program you are debugging. You can change the contents of these registers at any time, from the Bugbyter Command Level.

To change the contents of a 65Ø2 register from the Bugbyter command level, type the name of the register followed by an equal sign (=), type the value to be stored in the register, and press RETURN. For example, to set the contents of the A-register to $8D, type

```
A=8D
```

and press RETURN. This new value is immediately reflected under the A-register label in the Register subdisplay. In the same way, you can change the C-, R-, PC-, A-, X-, Y-, S-, and P-registers to any value you wish.

Bugbyter's "B" flag is not a register, and you cannot change this flag by using this register-assignment command. The NV-BDIZC display is just the binary representation of the P-register. To change individual bits in the P-register, you must type P= followed by the appropriate hexadecimal number.

WARNING
Bugbyter uses the first 32 bytes of the 65Ø2's stack (locations $1ØØ to $11F), and any attempt by you or your program to alter the stack pointer to point into this region could result in a conflict between Bugbyter and the program you are debugging.

Bugbyter flashes a warning in the Stack subdisplay area if you or your program sets the stack pointer to any value less than $2Ø (because the stack pointer points only to locations in Page One memory, a stack pointer value of $2Ø implies a memory address of $12Ø).

Decimal-to-Hexadecimal and Hexadecimal-to-Decimal Conversions: To help you when you assign numeric values to memory locations or registers, or whenever you convert numbers from one base to another, Bugbyter converts from hexadecimal to decimal, or from decimal to hexadecimal. For example, from the Bugbyter Command Level, if you type

```
$C3=
```

and press RETURN, Bugbyter converts the hexadecimal number $C3 to its decimal equivalent and displays this value following the equal sign:

```
$C3=ØØ195
```

In the same way, you can type hexadecimal numbers without the dollar sign ($). For example, if you type

```
C3=
```

and press RETURN, Bugbyter treats this command the same way it did the $C3= command above.

To convert a decimal number to its hexadecimal equivalent, precede the decimal number by a plus (+) or minus (-) sign, to distinguish it from a hexadecimal number. For example, if you type

```
+43=
```

and press RETURN, Bugbyter responds by displaying the hexadecimal equivalent:

```
+43=$2B
```

Altering Bugbyter's Master Display Layout (SET)

Bugbyter's Master Display has several subdisplays that let you view many different items of information at once. You can customize Bugbyter's Master Display layout to alter the relative sizes of some of these subdisplays. The SET command lets you do the following:

- Set more or fewer program breakpoints (in the Breakpoint subdisplay).

- View larger or smaller portions of your disassembled program (in the Code Disassembly subdisplay).

- View larger or smaller regions of the memory stack (n the Stack subdisplay).

- View more or fewer memory cells (in the Memory subdisplay).

To change the relative sizes of these subdisplays, from the Bugbyter Command Level, type

SET

and press RETURN. Bugbyter displays a sketch of the Code Disassembly and the Breakpoint subdisplays, and allows you to set the relative sizes of these displays.

Now you have two options:

- Use the LEFT ARROW key to increase the number of breakpoints that are displayed, and simultaneously decrease the size of the Code Disassembly subdisplay.

- Use the RIGHT ARROW key to decrease the number of breakpoints that are displayed, and simultaneously increase the size of the Code Disassembly subdisplay.

When you are satisfied with the relative sizes of these subdisplays, press RETURN to fix their relative sizes as you have selected them.

You can now select where Bugbyter displays the next-instruction-to-be-executed bar in the Code Disassembly subdisplay. Use the LEFT ARROW and RIGHT ARROW keys to adjust the position of the inverse-video bar within the Code Disassembly subdisplay. The position of this bar divides the subdisplay among the instructions that have been executed (above the bar) and instructions not yet executed (at and below the bar) when you are tracing or single-stepping your program. Press RETURN when you are satisfied with the bar's position.

Bugbyter then displays a sketch of the Stack and Memory Cell subdisplays. Use the LEFT ARROW and RIGHT ARROW keys to adjust the relative sizes of the Stack and Memory subdisplays, just as you did for the Disassembly and Breakpoint subdisplays.

When you are satisfied with the relative sizes of these displays, press RETURN to fix these sizes. Then use the LEFT ARROW and RIGHT ARROW keys to adjust the position of the stack-pointer bar within the Stack subdisplay. Typically, you will want this bar near the top of the subdisplay, as information on the stack appears below this inverse-video bar.

When you are finished, press RETURN to return to the Bugbyter Command Level.

When you use the SET command, you do not change the contents of the Memory subdisplay or the Breakpoint subdisplay. You can recover any Memory cells or Breakpoints that are no longer displayed by using the SET command to enlarge these subdisplays and display this information once again.

Controlling the Execution of Your Program

There are as many ways to debug a program as there are ways to write a program; both are a matter of the programmer's style. This manual is not intended to teach a particular way to debug your programs any more than it is intended to teach you a programming style. There are, however, several techniques for controlling the execution of your program that are very useful for debugging. This section discusses some of the techniques that you can use with Bugbyter. They include

- Single-stepping a program
- Tracing a program
- Executing a program directly on the 65Ø2.

Using Single-Step and Trace Modes

Bugbyter's Single-Step and Trace modes provide a powerful debugging environment. Bugbyter can trace practically any executable 65Ø2 program, including interrupt- and timing-sensitive code. If you followed the tutorial at the beginning of this chapter, you have already found how easy it is to use the Single-Step and Trace modes.

When you enter Single-Step or Trace mode, Bugbyter removes the usual command prompt from the command line, and replaces it with the words SINGLE STEP or TRACE. Once you enter one of these modes, you can no longer type commands at the Bugbyter Command Level. Instead, you can type a set of single-keystroke commands that give you access to a variety of debugging features.

These commands are summarized in Table 4-1. To access any of these debugging functions, you need only type the single keystroke.

Table 4-1. Debugging Commands in Single-Step and Trace Modes

Trace or Single-Step Operation	Keystroke
Return to the Bugbyter Command Level	ESC
Single-step (execute) one instruction	SPACE bar
Skip the next instruction	RIGHT ARROW
Trace the program until a breakpoint or end	RETURN
Trace until an RTS instruction is encountered	R
Clear the Cycle Count register	C
Use Paddle Ø to adjust trace rate	P
Use Keyboard Rate to adjust trace rate (set R=value before entering Trace mode)	K
Turn off Bugbyter sound (Quiet)	Q
Turn on Bugbyter sound	S
Display primary Apple screen	1
Display secondary Apple screen	2
Display the Apple text screen	T
Display the Apple low-res graphics screen	L
Display the Apple high-res graphics screen	H
Display the Apple full screen graphics	F
Display a mixed graphics screen, with the Bugbyter command line visible	M

To reenter Single-Step mode from the Bugbyter Command Level, type S and press RETURN. To enter Trace mode, type T and press RETURN. Bugbyter remembers the last instruction it executed before leaving Trace or Single-Step mode, and continues from the next instruction whenever you reenter Trace or Single-Step mode without specifying a starting address.

Single-Stepping Your Program

When you debug a program, you spend much of your time observing the execution of the program and verifying that it works correctly. Bugbyter's Single-Step mode (introduced in the tutorial), allows your Apple II to execute a single instruction at a time, stopping after each instruction so that you can observe the result.

To enter Single-Step mode from the Bugbyter Command Level, type the memory address of the first instruction that you wish to execute, follow it with an S, and press RETURN. In the the tutorial, you typed

```
1000S
```

After you type this Single-Step command, Bugbyter shows, in its Code Disassembly subdisplay, the first instruction that is to be executed. To execute this instruction, press the SPACE bar. Bugbyter executes this single instruction and updates the screen to show you the effect of the execution of that instruction.

To continue executing instructions, press the SPACE bar for each instruction to be executed. After Bugbyter executes each instruction, it updates its Master Display, allowing you to verify that your program has executed correctly, or showing you exactly where your program has gone wrong.

Using Trace Mode to Trace Subroutines

Although Bugbyter's Single-Step mode is handy for testing short programs (or small portions of larger programs), single-stepping through medium to large programs is rather tedious.

Most good programs contain subroutines. Bugbyter's Trace mode lets you step through a program one subroutine at a time. You can interrupt execution at places you specify, letting you observe program results and the status of registers and the stack. You can skip over portions of your program that are operating correctly, getting quickly to the location of a problem.

You can enter Trace mode only from Single-Step mode. To trace your program until the next occurrence of an RTS instruction (that is, until the end of the next subroutine), type

```
R
```

while you are in Single-Step mode. Bugbyter begins executing each instruction of your program in sequence, updating the Master display after each instruction, until it encounters either an RTS instruction or a breakpoint (breakpoints are discussed below). When Bugbyter encounters an RTS instruction, it returns you to Single-Step mode and lets you observe the results of the subroutine and continue debugging as you wish.

Setting Transparent Breakpoints

Another way to control Bugbyter's trace of your program is to set breakpoints at specified locations within your program.

When you are debugging in Trace or Single-Step mode, Bugbyter monitors your Apple II's program counter (PC) before executing each instruction, and compares it to any breakpoints that you may have set in the Breakpoint subdisplay. If your program reaches one of these breakpoints (that is, if the 65Ø2's program counter matches the address of a breakpoint), Bugbyter interrupts your program and returns you to Bugbyter's Command Level.

A **transparent breakpoint** does not alter your program's code in any way. **Real breakpoints** (used in Execution mode) alter your program temporarily.

If you didn't set any breakpoints, or if Bugbyter doesn't encounter any while executing your program, Bugbyter continues executing your program until you press the ESC key, or Bugbyter reaches the end of your program.

To set transparent breakpoints, you should be familiar with the Breakpoint subdisplay area of Bugbyter's Master Display:

```
-------------------------------------------------------------

              BP  POINT COUNT TRIG  BROKE
              1   ØØØØ  ØØØØ  ØØØØ  ØØØØ
              2   ØØØØ  ØØØØ  ØØØØ  ØØØØ
              3   ØØØØ  ØØØØ  ØØØØ  ØØØØ
              4   ØØØØ  ØØØØ  ØØØØ  ØØØØ

-------------------------------------------------------------
```

The Breakpoint subdisplay (in the lower right section of Bugbyter's Master Display) comprises several fields. You can use the SET command (described earlier) to increase or decrease the number of breakpoints displayed. The Breakpoint subdisplay has four column headings; each line under these four headings represent one breakpoint.

- POINT is the hexadecimal address of the breakpoint.

- COUNT is the number of times Bugbyter has encountered this breakpoint address while executing your program, since Bugbyter last TRIGgered, or interrupted program execution, at this breakpoint.

- TRIG is a count that you can specify. Bugbyter does not interrupt program execution until it has encountered this particular breakpoint the number of times that you specify. A TRIG count of 1 causes Bugbyter to interrupt program execution the first time it encounters this breakpoint.

- BROKE is the number of times Bugbyter has actually TRIGgered at this particular breakpoint.

To enter a breakpoint address from the Bugbyter Command Level, type BP followed by a breakpoint number. For example, to set a breakpoint to be displayed in row number 1 of the Breakpoint subdisplay, type

BP1

and press RETURN.

When you type this breakpoint set-up command, Bugbyter moves the cursor to the first zero in the POINT field of the breakpoint row you specify. Enter a hexadecimal address in this field to represent the address of breakpoint #1. Use the RIGHT ARROW key to move the cursor to the TRIG field on that same breakpoint line. In this TRIG field, type a hexadecimal number greater than zero. If you want Bugbyter to stop on the first occurrence of this breakpoint, type 1 in this field. (If you leave TRIG set to Ø, Bugbyter ignores this breakpoint.) Press RETURN when you have finished setting the breakpoint address and the TRIG count.

Where to set breakpoints? You might set breakpoints at critical locations in your program, such as just after a call to an important subroutine. When you trace the program, you can verify that the results from this subroutine are correct and are stored in the proper registers or memory locations. Other locations might be right after an important compare instruction, or anywhere you want to check the status of your program.

Using Breakpoints

Having set breakpoints where you want them, you can use Trace or Single-Step mode to monitor the execution of your program. Every time Bugbyter encounters an instruction located at a breakpoint address (POINT), Bugbyter increments the COUNT for that breakpoint and compares this COUNT to the TRIG value you set for that breakpoint. When the COUNT equals your TRIG value, Bugbyter stops your program execution before executing the instruction at the POINT address. Bugbyter then highlights the row in the Breakpoint subdisplay that corresponds to the breakpoint that TRIGgered, and clears the COUNT. You can then observe all of the 65Ø2 registers, stack, and other conditions that existed just before your program was interrupted by the breakpoint.

To continue executing the program from the point where the breakpoint occurred, type either T (to reenter Trace Mode) or S (to enter Single-Step mode), and press RETURN.

Clearing Transparent Breakpoints

To clear a particular breakpoint, type CLR followed by the breakpoint number, and press RETURN.

To clear all breakpoints, type CLR and press RETURN.

Adjusting the Trace Rate

During tracing, Bugbyter interprets each 65Ø2 instruction in your program. In other words, the Apple's 65Ø2 microprocessor executes the Bugbyter program, which in turn executes your program code. As a result, the code you are tracing executes much slower than if it were executed directly by the 65Ø2. There are three ways to adjust the rate at which Bugbyter traces your code.

- Before entering Trace mode, set a value in Bugbyter's Trace Rate register, displayed in the Register subdisplay. Type R= followed by a hexadecimal value from Ø to FF (Ø is the fastest rate and FF the slowest). Press RETURN. When you enter Trace mode, Bugbyter uses this rate setting to control the speed of its trace.

- Before entering Trace mode, type OFF from the Bugbyter Command Level and press RETURN. This clears Bugbyter's Master Display, greatly increasing the speed of the trace.

- If you have game paddles for your Apple II, you can use them to adjust the trace rate while you are in Trace mode. Type P and use Paddle Ø to adjust the trace rate. To disable the paddle and return to the keyboard (R register) rate, type K.

To restore the Master Display from Bugbyter's Command Level, type ON and press RETURN. To restore the Master Display from Bugbyter's Trace mode, press ESC, type ON, and press RETURN.

Using Display Options in Trace and Single-Step Modes

You can select one of seven display options for Bugbyter to use in Trace and Single-Step modes to display different types of relevant information. The option you select is displayed at the right edge of the Bugbyter Code Disassembly subdisplay, just to the right of the instruction to which these options refer.

You must select these options from the Bugbyter Command Level before you enter Trace or Single-Step mode. You can select only one option at a time. Bugbyter displays only the most recently selected option.

The following table shows the display options that you can select and the commands that select them.

Table 4-2. Display Options in Trace and Single-Step Modes

Display Option	Command (followed by **RETURN**)
To show the A-register in binary:	0=A
To show the X-register in binary:	0=X
To show the Y-register in binary:	0=Y
To show the Stack Pointer in binary:	0=S
To show the Processor Status Byte in binary:	0=P
To display the machine-instruction bytes in hex:	0=B
To show computed effective addresses, relative branches, and instruction cycles:	0=E

The last option shown in this table, the 0=E display option, requires some extra discussion. There are four 65Ø2 addressing modes for which the 65Ø2 internally computes an effective address. They are:

mode	**example**
indexed	LDA $3ØØ,X
indirect	JMP ($3ØØ)
indexed indirect	LDA ($1Ø,X)
indirect indexed	LDA ($1Ø),Y

In Trace or Single-Step mode, Bugbyter computes the actual or effective address for each instruction, using the current contents of registers or memory cells at the time the instruction is executed. If you selected the 0=E option, Bugbyter displays these effective addresses in the Disassembly subdisplay. Bugbyter also displays all relative branch offsets (the hex byte operand of a branch opcode) when you select this option.

When you select the 0=E option, Bugbyter also displays the number of instruction cycles used by each instruction. This count of the number of cycles appears in parentheses to the right of the Disassembly line for each instruction that is executed. For example:

C=Ø clears the Cycle Count register in the Register subdisplay.

O=E sets the display option to E.

A=12 sets the accumulator to $12.

FCA8S activates Single-Step mode at the start of the Monitor WAIT routine.

R starts Bugbyter tracing until it encounters an RTS instruction.

Bugbyter begins tracing the Monitor WAIT routine, with the command line at the bottom of the Master Display showing

```
TRACE                          AWAITING RTS
```

The rest of the Master Display changes rapidly. You can watch the Cycle Count register, in the upper left corner of the Master Display, incrementing after each instruction is executed. After a few seconds, the Master Display halts and shows the following:

```
-------------------------------------------------------------

  C     R  B  PC   A  X  Y  S  P  NV-BDIZC
  Ø41E  ØØ 0  FCB3 ØØ ØØ ØØ FF 33 ØØ11ØØ11

   1F9: 7C   FCAC: BNE $FCAA    E:  FC (2)
   1FA: 7C   FCAE: PLA          E:     (4)
   1FB: A1   FCAF: SBC #$Ø1     E:     (2)
   1FC: D2   FCB1: BNE $FCA9    E:  F6 (3)
   1FD: E3   FCA9: PHA          E:     (3)
   1FE: D6   FCAA: SBC #$Ø1     E:     (2)
   1FF: Ø1   FCAC: BNE $FCAA    E:  FC (2)
   1ØØ: FF   FCAE: PLA          E:     (4)
   1Ø1: FF   FCAF: SBC #$Ø1     E:     (2)
   1Ø2: ØØ   FCB1: BNE $FCA9    E:  F6 (2)
   1Ø3: ØØ   FCB3: RTS
   1Ø4: FF   FCB4: INC $42
   1Ø5: FF   FCB6: BNE $FCBA

  ØØØØ:4C L  BP  POINT COUNT TRIG  BROKE
  ØØØØ:4C L  1   ØØØØ  ØØØØ  ØØØØ  ØØØØ
  ØØØØ:4C L  2   ØØØØ  ØØØØ  ØØØØ  ØØØØ
  ØØØØ:4C L  3   ØØØØ  ØØØØ  ØØØØ  ØØØØ
  ØØØØ:4C L  4   ØØØØ  ØØØØ  ØØØØ  ØØØØ

  SINGLE STEP

-------------------------------------------------------------
```

The Cycle Count register in the upper left corner shows $41E (decimal 1Ø54) CPU cycles, representing the time required to execute the WAIT routine when the accumulator is preset to $12. Using a cycle time of one microsecond, this cycle count represents a WAIT of approximately one millisecond (Ø.ØØ1 seconds).

Bugbyter increments the Cycle Count register only when you select the O=E display option before entering Trace or Single-Step mode. The Cycle Count register is not incremented if you use the OFF command to turn off the Bugbyter Master Display.

Using Execution Mode

Bugbyter's Execution mode lets you execute portions of your program at the full speed of the 65Ø2 microprocessor, without having Bugbyter slow your program's execution by constantly checking for breakpoints.

Real Breakpoints

To debug your program in Execution mode, you must use real breakpoints, rather than transparent breakpoints, to control your program's execution.

A real breakpoint is a 65Ø2 BREAK opcode ($ØØ) that Bugbyter places at the breakpoint address in your program code. Although transparent breakpoints do not change your program code in any way, Bugbyter must alter your program code when it inserts real breakpoints.

To set Real breakpoints at the addresses shown in the POINT column of the Breakpoint subdisplay, type

IN

from Bugbyter's Command Level, and press RETURN. This command causes Bugbyter to insert 65Ø2 BRK opcodes (ØØ) at all of the breakpoint addresses (any POINT addresses with their associated TRIGs set to greater than zero). Bugbyter displays I for "breakpoints IN" under the B flag in the register subdisplay at the top of the screen.

Once you have inserted real breakpoints in your program code, you can debug your program while executing it at the full speed of the 65Ø2 microprocessor.

When real breakpoints are in your program, Bugbyter does not allow you to add, change, or clear any breakpoints. To modify any of the breakpoint information, you must first instruct Bugbyter to take real

breakpoints out. Type

OUT

from the Bugbyter Command Level and press RETURN. This command forces Bugbyter to remove the break opcodes that it inserted when you last used the IN command, restoring the 65Ø2 instruction opcodes originally stored there. The B flag that monitors the status of real breakpoints in the Register subdisplay at the top of the screen displays the letter O, for "breakpoints OUT."

WARNING
Once you have set real breakpoints in your program, Bugbyter has altered your program by inserting break opcodes at every breakpoint address. If you exit Bugbyter before you remove the real breakpoints, your code may be riddled with unwanted 65Ø2 breaks. Be sure to type

OUT

and press RETURN to return your code to its original condition.

If you enter Trace mode when real breakpoints are IN, Bugbyter operates just as it does when real breakpoints are OUT. It is more convenient to use transparent breakpoints if you wish to debug your program in Trace or Single-Step mode. Real breakpoints are necessary only when you are debugging your program in Execution mode.

Debugging Your Program in Execution Mode

To execute your program directly from the Bugbyter Command Level, type a starting address followed by G, and press RETURN (or simply type G and press RETURN).

When you type this command, Bugbyter enters Execution mode and starts executing your program from the specified starting address. If you didn't type a starting address, Bugbyter uses the last starting address that you specified with either the Single-Step, Trace, or Go command. For example, to execute your program starting at location $1ØØØ, type

1ØØØG

and press RETURN. Bugbyter treats this command in much the same way it treats the Apple Monitor G command: Bugbyter pushes a return address onto the stack before executing your program code. If your program encounters an RTS (return from subroutine) instruction, you will be

returned to the Bugbyter Command Level. The same result occurs if your program encounters a BRK opcode or a breakpoint.

To continue executing your program after a breakpoint has forced Bugbyter out of Execution mode, type J and press RETURN. This command does not push a return address on the stack, if you have already set up Execution mode by using the G command. When first executing your code, type

<starting-address>G

and press RETURN. After encountering any break opcodes, type J and press RETURN to continue direct, real-time execution.

By the Way: Because the J command does not push a return address on the stack, you should always begin Execution mode debugging by using the G command. If you start Execution mode with the J command and your program encounters an RTS instruction, the 65Ø2 uses an undefined address from the stack, with unpredictable results.

When the 65Ø2 encounters a BRK opcode (either a breakpoint inserted by Bugbyter or part of your program code itself), the 65Ø2 passes control back to the Bugbyter program. If the BRK opcode is a breakpoint inserted by Bugbyter, Bugbyter highlights that breakpoint (using inverse video in the Breakpoint subdisplay), and returns you to the Bugbyter Command Level. If the BRK opcode is not a breakpoint inserted by Bugbyter, Bugbyter passes control to the Apple Monitor.

Debugging Real-Time Code

If you are debugging a program that contains real-time sensitive code or calls real-time Apple II routines, these portions of your program will not function correctly if you merely simulate their execution in Bugbyter Simulation mode. There are two ways to use Bugbyter to debug this kind of program. One way was described in the section on using real breakpoints in Execution mode. The other way is to use the more powerful debugging capabilities of Bugbyter Simulation mode to debug the insensitive portions of your program, while still executing the real-time routines of your program at the full speed of the 65Ø2 microprocessor.

Examples of real-time sensitive code are the core routines associated with the ProDOS disk operating system. The read data, write data, read address, and track seek routines are very sensitive to cycle speed variation. These core routines will not function at all if you trace

them with Bugbyter's Simulation mode. To execute these subroutines from the program that you are tracing, you must indicate to Bugbyter that these are real-time portions of code.

Bugbyter allows you to execute subroutines in Execution mode, while tracing your calling routines in Simulation mode. To set up this method, designate a region of memory containing the real-time routines that the Bugbyter will always execute in Execution mode. Do this by setting two soft-switch locations within the Bugbyter program code.

Soft switches are memory locations at the start of the Bugbyter program code that you can set or clear to control features of the Bugbyter. Offset from the beginning of the Bugbyter program, at locations start+$ØA and start+$ØB, are the two bytes that you can set to define the starting address of this real-time region. At locations start+$ØC and start+$ØD are two bytes that indicate the ending address of this region.

When you are tracing your program in Simulation mode, any subroutine calls (JSRs) to locations inside this specified region will cause Bugbyter to transfer control of your program from Simulation mode to Execution mode. This means that the subroutines in this region will execute at the full speed of the 65Ø2. When the 65Ø2 encounters the return from subroutine (RTS) instruction that transfers control back to your calling routine, Bugbyter reactivates Simulation mode and continues tracing or single-stepping your main or calling program. For example, if you have Bugbyter loaded at $2ØØØ, typing

```
2ØØA:  Ø DØ FF FF
```

and pressing RETURN specifies a real-time region extending from location $DØØØ to $FFFF. If you then typed

```
3ØØ:LDX #Ø
3Ø2:LDA 11ØØ,X
3Ø5:JSR FDFØ     (the address of the Monitor character-out routine)
3Ø8:INX
3Ø9:CPX #CØ
3ØB:BNE 3ØØ
```

and

```
3ØC:BRK
```

you would set up a calling program to output the characters at $11ØØ-11BF. Then type

```
OFF                    (turns the Master Display off)
```

and

```
3ØØT                   (traces your calling program)
```

Bugbyter begins tracing this character output routine until it encounters a JSR to the Monitor ROM routine (called COUT1) at $FDFØ--inside our real-time region ($DØØØ-FFFF). At that time, Bugbyter enters Execution mode and allows COUT1 to execute directly under the 65Ø2. When the COUT1 routine exits back to the routine, Bugbyter reenters Simulation mode and the code that followed the call (JSR) to COUT1 then continues tracing under Simulation mode.

Debugging Programs That Use the Keyboard and Display

Bugbyter uses the display screen to send information to you, and many of Bugbyter's features require that you use the keyboard to enter debugging commands.

This section deals with potential contention, between your program and Bugbyter, for these resources. This contention between Bugbyter and your own program must be eliminated.

Eliminating Contention for the Screen

Because Bugbyter displays all information on the screen, using absolute screen addressing, any programmed output by your program that uses Apple's I/O hooks (CSW) will not be impeded. However, as Bugbyter uses the screen to constantly update the Master Display when you are in Trace or Single-Step mode, you are not likely to see any of your program's output.

To eliminate contention between Bugbyter and your program for the use of the screen, just type (from Bugbyter's Command Level):

```
OFF
```

and press RETURN. Bugbyter clears the first 23 lines of the text page, leaving the command prompt appearing on the bottom line. When you enter Trace or Single-Step mode, Bugbyter will allow your program to write anything it desires to this text page.

To recall the Master Display from the Bugbyter Command Level, simply type

```
ON
```

and press RETURN in response to the Bugbyter command prompt. If you are currently in Trace or Single-Step mode, first press ESC, to exit to the Bugbyter Command Level.

Eliminating Contention for the Keyboard

When your program requires input from the keyboard, you should make sure that Bugbyter does not interfere with your program's polling of the keyboard. Normally in Trace or Single-Step mode, Bugbyter samples (polls) the keyboard to respond when you type one of the single-keystroke commands described previously. If you are tracing or single-stepping a program that expects input from the keyboard, your program will never receive any characters unless you turn off Bugbyter's keyboard polling.

When you turn off Bugbyter's keyboard polling, Bugbyter ignores all characters typed at the keyboard, except one special interrupt character that you specify. Bugbyter scans the keyboard address ($CØØØ) during Trace or Single-Step mode, but does not clear the keyboard ready flag unless the character you typed is this special interrupt character. For any other characters, Bugbyter leaves these addresses intact and permits your program to accept the keystroke from the keyboard input register.

To turn off Bugbyter's keyboard polling, set Bugbyter's keyboard-polling soft switch.

The Keyboard-Polling soft switch is located at memory location $7CØ6 (or the relative memory location "start + 6" if you have relocated Bugbyter to a different location). If you relocate Bugbyter to a different address, the keyboard-polling soft switch will be located at the sixth location following the start of the Bugbyter. The byte stored at this location consists of

- A one-bit keyboard-polling flag (the most significant bit)
- A seven-bit ASCII value that defines the special interrupt key.

When you first start Bugbyter, this most significant bit is Ø. If you set this keyboard-polling flag to 1, Bugbyter turns off its keyboard polling when in Trace or Single-Step mode. With keyboard polling turned off, Bugbyter continues to scan for the special interrupt keystroke that you specified. For example, typing

```
7CØ6:81
```

from the Bugbyter Command Level and pressing RETURN turns off Bugbyter's keyboard polling in Trace mode and instructs Bugbyter to scan for a CONTROL-A ($Ø1) character. The program you are debugging can then accept any keystroke from the keyboard, except the CONTROL-A character. When you type CONTROL-A in Trace mode, Bugbyter will exit from Trace mode and return you to the Bugbyter Command Level.

Using Paddle Button Ø to Control Trace Mode

Rather than having Bugbyter scan the keyboard during Trace mode, you can communicate with Bugbyter in Trace mode by using Paddle Button Ø. This technique frees the keyboard so that your program can accept any keyboard character as input.

By the Way: The OPEN APPLE key on the Apple IIe's keyboard functions exactly like Paddle Button Ø. If you have an Apple IIe, you can use Paddle Button Ø and the OPEN APPLE key interchangeably.

To use Paddle Button Ø to suspend Bugbyter's Trace mode, you must activate this feature before you enter Trace mode. Pressing Paddle Button Ø does not cause Bugbyter to exit Trace mode, as does pressing ESC or the "interrupt" key when Bugbyter's keyboard polling has been turned off. Instead, when this feature is activated and you are in Trace mode, Paddle Button Ø suspends Bugbyter's trace of your program. Bugbyter continues tracing as soon as Paddle Button Ø is released.

To set up this feature, set Bugbyter's Paddle-Button-Ø soft switch to $8Ø (normally, this soft switch is cleared or $ØØ). This soft switch is located at memory address 2ØØ4 (or "start + 4" if you have relocated Bugbyter to a different memory location). To set this soft switch, type

```
2ØØ4:8Ø
```

and press RETURN.

Note: If you set the Paddle-Button-Ø soft switch to $8Ø and you have not connected game paddle Ø to your Apple II's Game I/O port, your Apple will be "frozen" if you enter Bugbyter's Trace mode. This is because your Apple II sees a disconnected game paddle as a game paddle with the paddle button continuously depressed.

This will not happen if you are using an Apple IIe, however, because the OPEN APPLE key is a permanently connected "Paddle Button Ø."

Using Paddle Ø

To use Paddle Ø to control Bugbyter's trace rate when you are in Trace mode, type a P in Trace mode. To deactivate this feature and cause Bugbyter to resume using the keyboard trace rate (shown under the R label in the Register subdisplay), type K while you are in Trace mode.

If you are debugging a program that itself uses Paddle Ø, you may want to disable this feature of Bugbyter, causing Bugbyter to ignore a P keystroke when you are in Trace mode. To disable Bugbyter's use of Paddle Ø, you must clear Bugbyter's Paddle-Ø soft switch, located at memory address 2ØØ6 (or "start + 6" if you have relocated Bugbyter). To clear this soft switch, type

```
2ØØ6:Ø
```

and press RETURN.

By the Way: To have positive control over Bugbyter's use of all three of your Apple II's input devices, you can set or clear the Paddle-Button-Ø, Paddle-Ø, and the Keyboard-Polling soft switches all at once. To disable Bugbyter's use of the game paddles and to activate Bugbyter's keyboard polling, clear all three of these soft-switches at once by typing

```
2ØØ4:Ø Ø Ø
```

and pressing RETURN.

Executing Undefined Op-Codes

When you are debugging your program in Trace or Single-Step mode, Bugbyter ignores any illegal or undefined 65Ø2 instruction opcodes. You can disable this restriction by using a Bugbyter soft switch.

To allow Bugbyter to execute undefined 65Ø2 opcodes, you must set the byte at relative location start+3 (absolute location $2ØØ3 if the Bugbyter was loaded at address $2ØØØ) to $8Ø. (This byte is set to $ØØ when Bugbyter is first loaded.) To prevent Bugbyter from executing illegal 65Ø2 instruction opcodes, you must use Bugbyter's normal memory assignment commands to reset this location to $ØØ.

Because Bugbyter does not know the length of an undefined opcode's operand, Bugbyter assumes no operand and increments the 65Ø2 PC by one. You must use the RIGHT ARROW key in Single-Step mode to skip past any

operands to the next instruction. Using the complete register and memory display capabilities of Bugbyter, you can easily explore all of the undefined operations of the 65Ø2 (try executing AF 58 FF, for example).

Chapter 5

The Relocating Loader

Chapter 5

The Relocating Loader

About This Chapter

The Relocating Loader routines in the ProDOS Assembler Tools allow you to run assembly-language subroutines from within BASIC programs. Read this chapter if you intend to use assembly-language programs as part of an Applesoft BASIC program.

There are two ways to run an assembly-language subroutine from a BASIC program:

- Use the ProDOS BLOAD command to load your binary object file into memory. Then call your routine, using the fixed starting address at which your program was assembled. (For details on the BLOAD command, see BASIC Programming With ProDOS.)

- Use the Relocating Loader routines to load your relocatable object program just below the HIMEM address. This method is preferable, because it allows BASIC programs to run efficiently on Apple II systems with varying amounts of memory.

The first part of this chapter explains what the Relocating Loader routines are and what they do.

The second part of this chapter explains how you can use these routines from within a BASIC program.

Overview

The Relocating Loader consists of two routines: RBOOT and RLOAD. Together, these routines allow your BASIC programs to load relocatable assembly-language subroutines into high memory, without regard to the amount of memory in a particular Apple II system. After your BASIC program has loaded your assembly-language subroutines, your BASIC program can call these subroutines at any time.

By the Way: The Relocating Loader routines load only relocatable assembly-language modules (REL files) that you generate using the Assembler's REL directive. The REL assembly directive is explained in Chapter 3.

To use the Relocating Loader, your BASIC program must first BRUN the RBOOT routine from BASIC. The RBOOT routine loads and sets up the RLOAD routine, which relocates the assembly-language modules. Once you have called the RBOOT routine, your BASIC program can call the RLOAD routine to load your individual relocatable subroutines into memory. After loading each relocatable subroutine into memory, the RLOAD routine returns an address at which your BASIC program can later call the actual assembly-language subroutine to perform its particular function.

When you invoke the RLOAD routine to load your assembly-language modules, RLOAD loads the designated module just below the HIMEM address of your Apple II system. RLOAD then reduces HIMEM in increments of 256 bytes, using the BASIC routine GETBUFR (described in PRODOS Technical Note #9), protecting the relocatable module from being overwritten by BASIC. You can load as many relocatable modules in this fashion as you like, but each call to RLOAD will allocate a minimum of 256 bytes per module.

The Relocating Loader routines do not constitute a linking loader. Although you can use these routines to load as many assembly-language modules as you like, the Relocating Loader does not resolve inter-module references or external symbols that you may have defined using the Assembler's EXTRN or ENTRY pseudo-op directives. Typically, if you use more than one assembly-language module in your BASIC program, you will have to call each module separately from Applesoft.

Restrictions

Although you can use the Relocating Loader routines with almost any BASIC program, there are restrictions on how and when you can use these routines:

- Before calling the Relocating Loader, your program should not allocate or use any string variables. The RLOAD routine does not save any string data that your program may have allocated before RLOAD loads relocatable modules into memory. (Numeric variables can be used before your program calls the Relocating Loader.)

- After calling RBOOT, your program must not DIMENSION or allocate any new numeric or string variables until after the last use of RLOAD to pull in your relocatable modules. Any variables that you use during this process must have been allocated before your program called RBOOT. This restriction is necessary because the RBOOT and RLOAD routines occupy memory just above Applesoft's variable tables. After you have used RLOAD for the last time, you are free to allocate new BASIC variables that will overwrite the RLOAD routine and reuse this memory space.

- RLOAD calls GETBUFR to allocate space for the REL modules, but it does not set the bits in the PRODOS memory map to protect the module's memory space. Each individual module must set the PRODOS memory map bits according to its own requirements.

In addition, RLOAD frees the memory map bits and the memory space and returns it to ProDOS for other uses. Each program using RLOAD must clear the memory map bits and call the BASIC FREEBUFR routine at $BEF8 (CALL 48888 from BASIC) before terminating.

Using the Relocating Loader

To invoke the Relocating Loader from a BASIC program, your program must first load and execute the RBOOT routine from Applesoft BASIC.

The following example shows the syntax of the BASIC statements that invoke the Relocating Loader routines and load a relocatable module called MYMODULE from disk:

```
10 ADRS = 0 : REM PRE-ALLOCATE ADRS VARIABLE

20 PRINT CHR$(4);"BRUN RBOOT" : REM LOAD AND EXECUTE RBOOT

30 ADRS=USR(0),"MYMODULE" : REM LOAD A RELOCATABLE MODULE
```

The RBOOT routine is a small subroutine that occupies memory from $218 through $3CF.

Note that you cannot use the usual D$ for the ProDOS BLOAD command; instead use a CHR($4), because you cannot allocate string variables before using the RBOOT and RLOAD routines.

When you call RBOOT to load the RLOAD routine into memory, RBOOT loads the RLOAD routine above the end of Applesoft's variable tables. RBOOT accepts no parameters, but assumes that the RLOAD function is on the disk that was last accessed, which typically would be the disk from which RBOOT itself was just BLOADed. The RBOOT routine then sets up the USR(0) function with the starting address of RLOAD.

RBOOT always looks for the name RLOAD. RBOOT uses the current prefix if it is set, otherwise RBOOT looks in the root directory of the last accessed disk device to locate RLOAD. The last accessed disk device will be the disk containing RBOOT. Refer to the ProDOS User's Manual for the exact description of the BRUN command and its behavior in locating RBOOT.

Your BASIC program can then invoke the RLOAD routine by calling the USR(0) function to load relocatable modules.

The RLOAD function takes one parameter from the statement containing the USR(0) function: The ProDOS pathname of the module to be loaded. The pathname must be the name of a REL type file, not a BINARY type file. If the pathname is omitted, the error message FILE NOT FOUND is displayed.

The RLOAD routine either returns the load address of the relocatable module that you have loaded, or returns an error if it encounters a problem while attempting to load your module. You can catch this error by using Applesoft's ON ERR facility. If you do not use the ON ERR statement before using RBOOT and RLOAD, and RLOAD encounters an error

while trying to load your module, your program will not function correctly.

The value returned by the USR function is a signed REAL result that your program can later use to CALL the loaded module. (This assumes that your relocatable module begins with an executable code segment.)

An effective means of providing multiple entry points to your relocatable module is to put a table of jump instructions at the start of the module. This allows your BASIC program to execute CALLs to the returned ADRS, ADRS+3, ADRS+6, ADRS+9, and so forth, as a means of entering the various subfunctions in your module. This technique also allows the contents of your relocatable module to grow or shrink later without disturbing your BASIC program's interface to the assembly-language module. You can later add additional jumps to the table for new functions, while leaving undisturbed the existing interface to your original entry points.

You can load as many modules as you wish, up to the available memory space minus the size of RLOAD itself. RLOAD is about 1.5K and is always loaded on a page boundary by RBOOT. RLOAD requires at least one free file buffer available that it can borrow from ProDOS. If it does not find one, the error message NO BUFFERS AVAILABLE appears.

By the Way: Because RLOAD reduces the address of HIMEM when it loads a module, your program must restore HIMEM and the ProDOS memory map by calling FREEBUFR as described earlier in this section. The Relocating Loader does not automatically restore HIMEM to its original setting.

When you test a program that uses the Relocating Loader, repeated use of RLOAD without restoring HIMEM causes RLOAD to allocate new space each time it loads a module. This can quickly consume all of memory if you do not restore HIMEM to its normal value before each test.

Appendixes

Contents

Appendixes

Contents

Appendix A

Quick Reference Guide to the Editor

Editor Commands Arranged by Function

Accessing Disk Volumes and Directories

CAT [pathname]	Display the 4Ø-column ProDOS catalog, using the current prefix or the optional pathname.
CATALOG [pathname]	Display the 8Ø-column ProDOS catalog, using the current prefix or the optional pathname.
Online	Display a list of the volume names of all mounted disks.
PreFiX [pathname]	Set the ProDOS prefix to the optional pathname. If the pathname is omitted, display the current prefix.
PreFiX /	If used alone, display the current prefix. If followed by a new prefix, change the prefix.

Storing and Retrieving Text Files

LOaD pathname	Load the file specified by pathname into the text buffer, destroying anything already in the buffer.
APPEND [line#] pathname	Replace [line#] and lines following it with specified file.
SaVE [begin# [-end#]] [pathname]	Save the lines in the specified range to the file specified by pathname.
FILE	Display the pathname of the file currently loaded, and the bytes used and bytes remaining in the buffer.

Manipulating Lines in the Text Buffer

Add [line#]	Add lines, starting with line#, to text buffer. Enter Input mode to add the new lines.
Insert line#	Insert lines from the keyboard before line#. Enter Input mode to insert the new lines.
COpy line#1 [-line#2] TO line#3	Insert line#1 [through line#2] before line#3.
Delete begin# [-end#]	Delete the specified range of lines from the buffer.
Replace begin# [-end#]	Delete the specified line or range of lines and enter Input mode, to add new lines to the text buffer in place of the deleted lines.
NEW	Delete (via pointer clearing) the entire contents of the text buffer.

Viewing the Text in the Text Buffer

List [begin# [-end#]]	Display, with line numbers, the lines in the specified range. Control characters displayed in inverse video.
CONTROL-R	Repeat the most recent List command.
Print [begin# [-end#]]	Display, without line numbers, the lines within the specified range. Control characters are displayed as control characters.

Changing Text Within a Line

Find [begin# [-end#]] [.string.]	Display all lines, within the specified range, that contain the string.
Change [begin# [-end#]] .oldstr.newstring.	Change all or some of the occurrences of a string to a new string within the specified range.
Edit [begin# [-end#]] [.string.]	Enter Edit mode for all lines that contain the specified string.
SET Delimchar	Change the Editor command delimiter character to the specified new delimiter character. (Permits you to use the colon in a search string.)

Editing Two Files at Once

SWAP	Swap the currently active text buffer, saving the contents of the current one for later editing.
KILL2	Delete the entire contents of text buffer 2 and return to single-buffer mode with text buffer 1.

Altering the Display

COLumn 4Ø	Set the Editor display routines to 4Ø-column display mode.
COLumn 8Ø	Set the Editor display routines to 8Ø-column display mode if your Apple II can support this.
CONTROL-E	Enable lowercase character entry (Apple II or Apple II Plus systems that cannot otherwise receive lowercase characters).
CONTROL-W	Disable lowercase character entry that was enabled by CONTROL-E.
SET Lcase	Enable lowercase character entry (for Apple II or II Plus systems that do not contain an ALS Smarterm Card, but can receive and display lowercase characters).
SET Ucase	Disable lowercase character entry that was enabled by the SET Lcase command.
Tabs [Tablist] [.tabchar.]	Set the Editor display tab stops to the Tablist, and set the tab character to tabchar.
TRuncON	Turn on truncation of comment display.
TRuncOFf	Turn off comment display truncation.

Leaving the Editor

END	End the Editor session and returns control to the BASIC interpreter without ProDOS.
EXIT [pathname]	Exit from the Editor to BAS.SYSTEM or to the optional pathname.
MON	Enter the Apple II Monitor. (Press CONTROL-Y to return to the Editor.)

Where line#	Display the hexadecimal address of text for the specified line# (the address useful when you're using Monitor commands).

Loading and Saving Non-Text Files

BLOAD pathname,A[$]address	Load the specified binary file into the edit buffer at the specified address. Anything already in the buffer may be destroyed.
BSAVE pathname,A[$]address, L[$]length	Save a binary image of the specified number of bytes, starting at the specified address, into a binary file with the specified pathname.
XLOAD pathname[,A[$]address]	Load a sequential file into the edit buffer, reconstructing its ProDOS filetype, access, and auxtype. Begin loading the data at the optional address if given.
XSAVE pathname [,A[$]address [,L[$]length]]	Save a sequential file from the edit buffer and reconstructs its filetype, access, and auxtype. Uses the optional address and length if they are given.

Managing Disk Directories

CREATE pathname	Create a subdirectory file with the specified pathname. The prefix applies in the normal manner.
DELETE pathname	Delete the file or subdirectory specified by the prefix and/or pathname.
RENAME oldpathname,newpathname	Change the name of an existing unlocked file from oldpathname to newpathname.
LOCK pathname	Lock the specified file.

UNLOCK pathname	Unlock the specified locked file.

Printing Files

PR# slot# [,DevInitstrg]	Set the Editor and Assembler printer device to slot# and save the DevInitstrg for device initialization.
PTROFF	Disable printer output.
PTRON	Enable printer output.

Automatic Command Execution

EXEC pathname	Begin reading Editor commands from the specified sequential text file.

Invoking the Assembler

PR# slot# [,DevInitstrg]	Set the Editor and Assembler printer device to slot# and saves the DevInitstrg for device initialization.
PR# diskslot,[Pnn] [Lnn]pathname	Select disk output for the Assembler listing and writes it into the specified text file. The pathname may not be a partial pathname if it begins with P or L.
ASM pathname[,objpathname]	Invoke the Assembler, to assemble the source file (pathname) and create a named object file (objpathname) or to suppress the disk object file (@).

Editor Commands Arranged Alphabetically

Add [line#]	Add lines, starting with line#, to text buffer. Enter Input mode to add the new lines.
APPEND [line#] pathname	Replace [line#] and lines following it with specified file.
ASM pathname[,objpathname]	Invoke the Assembler, to assemble the source file (pathname) and create a named object file (objpathname) or to suppress the disk object file (@).
BLOAD pathname,A[$]address	Load the specified binary file into the edit buffer at the specified address. Anything already in the buffer may be destroyed.
BSAVE pathname,A[$]address, L[$]length	Save a binary image of the specified number of bytes, starting at the specified address, into a binary file with the specified pathname.
CAT [pathname]	Display the 40-column ProDOS catalog, using the current prefix or the optional pathname.
CATALOG [pathname]	Display the 80-column ProDOS catalog, using the current prefix or the optional pathname.
COLumn 40	Set the Editor display routines to 40-column display mode.
COLumn 80	Set the Editor display routines to 80-column display mode, if your Apple II can support this.
COpy line#1 [-line#2] TO line#3	Insert line#1 [through line#2] before line#3.
CREATE pathname	Create a subdirectory file with the specified pathname. The prefix applies in the normal manner.

Change [begin# [-end#]] .oldstr.newstring.	Change all or some of the occurrences of a string to a new string within the specified range.
DELETE pathname	Delete the file or subdirectory specified by the prefix and/or pathname.
Del begin# [-end#]	Delete the range of lines from the buffer.
Edit [begin# [-end#]] [.string.]	Enter Edit mode for all lines that contain the specified string.
END	End the Editor session and returns control to the BASIC interpreter without ProDOS.
EXEC pathname	Begin reading Editor commands from the specified sequential text file.
EXIT [pathname]	Exit from the Editor to BAS.SYSTEM or to the optional pathname.
FILE	Display the pathname of the file currently loaded, and the bytes used and bytes remaining in the buffer.
Find [begin# [-end#]] [.string.]	Display all lines, within the specified range, that contain the string.
Insert line#	Insert lines from the keyboard before line#. Enter Input mode to insert the new lines.
KILL2	Delete the entire contents of text buffer 2 and return to single-buffer mode with text buffer 1.
List [begin# [-end#]]	Display, with line numbers, the lines in the specified range. Control characters displayed in inverse video.
LOCK pathname	Lock the specified file.

LOaD pathname	Load the file specified by pathname into the text buffer, destroying anything already in the buffer.
MON	Enter the Apple II Monitor. (Press CONTROL-Y to return to the Editor.)
NEW	Delete (via pointer clearing) the entire contents of the text buffer.
Online	Display a list of the volume names of all mounted disks.
PreFiX [pathname]	If used alone, display the current prefix. If followed by a new prefix, change the prefix.
PreFiX /	Display current ProDOS prefix.
Print [begin# [-end#]]	Display, without line numbers, the lines within the specified range. Control characters are displayed as control characters.
PR# slot# [,DevInitstrg]	Set the Editor and Assembler printer device to slot# and saves the DevInitstrg for device initialization.
PR# diskslot,[Pnn] [Lnn]pathname	Select disk output for the Assembler listing and writes it into the specified text file. The pathname may not be a partial pathname if it begins with P or L.
PTROFF	Disable printer output.
PTRON	Enable printer output.
RENAME oldpathname,newpathname	Change the name of an existing unlocked file from oldpathname to newpathname.

Replace begin# [-end#]	Delete the specified line or range of lines and enters Input mode, to add new lines to the text buffer in place of the deleted lines.
SaVE [begin# [-end#]] [pathname]	Save the lines in the specified range to the file specified by pathname.
SET Delimchar	Change the Editor command delimiter character to the specified new delimiter character. (Permits you to use the colon in a search string.)
SET Lcase	Enable lowercase character entry (for Apple II or II Plus systems that do not contain an ALS Smarterm Card, but can receive and display lowercase characters).
SET Ucase	Disable lowercase character entry that was enabled by the SET Lcase command.
SWAP	Swap the currently active text buffer, saving the contents of the current one for later editing.
Tabs [Tablist] [.tabchar.]	Set the Editor display tab stops to the Tablist, and set the tab character to tabchar.
TRuncON	Turn on truncation of comment display.
TRuncOFf	Turn off comment display truncation.
UNLOCK pathname	Unlock the specified locked file.
Where line#	Display the hexadecimal address of text for the specified line# (the address useful when you're using Monitor commands).

XLOAD pathname[,A[$]address]	Load a sequential file into the edit buffer, reconstructing its ProDOS filetype, access, and auxtype. Begin loading the data at the optional address if given.
XSAVE pathname [,A[$]address [,L[$]length]]	Save a sequential file from the edit buffer and reconstruct its filetype, access, and auxtype. Use the optional address and length if they are given.

Edit Mode Keystroke Summary

Editing Function	Keystroke
Move cursor **left** one character	LEFT ARROW
Move cursor **right** one character	RIGHT ARROW
Delete current character	CONTROL-D
Insert character(s) at cursor position	CONTROL-I
Replace character(s)	any noncontrol character
Accept next character **verbatim**	CONTROL-V
Restore the original line	CONTROL-R
Find character, move cursor to it	CONTROL-F, any character
Store line as it appears on the screen	RETURN
Truncate line at cursor, moving truncated portion to text buffer	CONTROL-T
Exit from Edit mode, return to Command Level	CONTROL-X
Enable lowercase input (if this was set up before Edit mode was entered)	CONTROL-E
Disable lowercase input (if this was set up before Edit mode was entered)	CONTROL-W

Appendix B

Quick Guide to 65Ø2 Assembly Language

Note: This is only a summary. For complete details, refer to one of the 65Ø2 programming manuals listed in the Preface.

Summary of Addressing Modes

Note that all required syntax may be preceded by an optional identifier in the label field of a statement.

Table B-1.
Summary of Addressing Modes

Addressing Mode	**Required Syntax**
Implied (no address)	opc
Accumulator	opc A
Immediate	opc #expression
Low 8 bits of address	opc #>expression
High 8 bits of address	opc #<expression
Zero page	opc zpg-expression
Indexed X	opc zpg-expression,X
Indexed Y	opc zpg-expression,Y
Absolute	opc abs-expression
Indexed X	opc abs-expression,X
Indexed Y	opc abs-expression,Y
Indexed,Indirect X	opc (zpg-expression,X)
Indirect,Indexed Y	opc (zpg-expression),Y
Absolute Indirect	JMP (abs-expression)

Where

- opc refers to an instruction mnemonic
- abs refers to an absolute address expression
- zpg refers to a zero page address expression.

All other characters must be typed as shown.

Note: To invoke a zero page wrap-around address calculation, you must use the low-byte numeric operator (>) in front of the expression.

Summary of Assembler Directives

Syntax	Description
ASC Dstring or .ASCII Dstring	ASCII character data
CHN filename[,[slot][,[drive][,vol]]]	CHaiN to new source file
CHR ?	CHaRacter used for REPeat
DATE	DATE character data (9)
DCI Dstring	Special ASCII character data
DDB expr[,expr...]	Define Double Byte
DEF identifier	DEFine absolute identifier
DEND	Dsect END
DFB expr[,expr...]	DeFine Byte(s)
DO expr	DO assembly if expr > Ø
DS expr[,expr]	Define Storage [value]
DSECT	Dummy SECTion beginning
DW expr[,expr...]	Define Word(s)
ELSE	Complement assembly mode
ENTRY identifier	Define ENTRY identifier
identifier EQU expr	EQUate identifier to expr
EXTRN identifier	Refer to EXTeRNal identifier
FAIL p,Dstring	FAILure error message
FIN	FINish conditional assembly
IBUFSIZ expr	Include BUFfer SIZe set
IDNUM	Generate TIME data (6 bytes)
IFEQ expr IFGE expr IFGT expr	Do assembly IF expr EQ Ø etc.
IFNE expr IFLE expr IFLT expr	Do assembly IF expr NE Ø etc.

INCLUDE filename[,[slot][,[drive][,vol]]]	INCLUDE filename in source
INTERP	Select INTERPreter object file
LST [ON , OFF][,[NO]opt[,[NO]opt...]]	LiSTing control and options
MACLIB [slot][,[drive][,vol]]	MACro LIBrary disk enable
MSB [ON , OFF]	Most Significant Bit set
OBJ expr	OBJect memory address set
ORG expr	ORiGin of assembler adrs
PAGE	Eject a PAGE on listing
REF identifier	REFerence global identifier
REL	Generate RELocatable output
REP expr	REPeat CHR expr times
SBTL Dstring	Define SuB TitLe string
SBUFSIZ expr	Source BUFfer SIZe set
SKP expr	SKiP expr blank lines
STR Dstring	Counted ASCII STRing data
SYS	Change filetype of next object file
TIME	Generate time word in output
ZDEF identifier	Zero page global DEFinition
ZREF identifier or ZXTRN identifier	Zero page eXTeRNal REFerence

Summary of 65Ø2 Mnemonics

Opcode	Effect
ADC	A + M + C -> A
AND	A and M -> A
ASL	C <- [7..Ø] <-Ø
BCC	Branch on C = Ø
BCS	Branch on C = 1
BEQ	Branch on Z = 1
BIT	A and M, M7 -> N, M6 -> V
BGE	Branch on C = 1
BLT	Branch on C = Ø
BMI	Branch on N = 1
BNE	Branch on Z = Ø
BPL	Branch on N = Ø
BRK	Force Break
BVC	Branch on V = Ø
BVS	Branch on V = 1
CLC	Ø -> C
CLD	Ø -> D
CLI	Ø -> I
CLV	Ø -> V
CMP	A - M status -> P
CPX	X - M status -> P
CPY	Y - M status -> P
DEC	M - 1 -> M
DEX	X - 1 -> X
DEY	Y - 1 -> Y
EOR	A xor M -> A
INC	M + 1 -> M
INX	X + 1 -> X
INY	Y + 1 -> Y
JMP	Jump to New Location
JSR	Jump to Subroutine
LDA	M -> A
LDX	M -> X
LDY	M -> Y
LSL	C <- [7..Ø] <- Ø
LSR	Ø -> [7...Ø] -> C
NOP	No Operation (PC=PC+1)
ORA	M or A -> A

```
PHA     A -> Ms     S-1 -> S
PHP     P -> Ms     S-1 -> S
PLA     S+1 -> S    Ms  -> A
PLP     S+1 -> S    Ms  -> P

ROL     -<- [7..Ø] <- C <-
ROR     --> C -> [7..Ø] ->-
RTI     Return from Interrupt
RTS     Return from Subroutine

SBC     A - M - C -> A
SEC     1 -> C
SED     1 -> D
SEI     1 -> I
STA     A -> M
STX     X -> M
STY     Y -> M

TAX     A -> X
TAY     A -> Y
TSX     S -> X
TXA     X -> A
TXS     X -> S
TYA     Y -> A
```

Note:

A, X, Y, S, and P are the 65Ø2 registers
M is a memory location
C is the Carry bit of the P-register
+ indicates binary addition
[7...Ø] is a bit description of M or A
Ms indicates the memory location pointed to by the S-register

---

Additional 65CØ2 Mnemonics

Instruction Mnemonics

Op Code	Description
BRA	Branch relative always [Relative]
DEA	Decrement accumulator [Accum]
INA	Increment accumulator [Accum]
PHX	Push X on stack [Implied]
PHY	Push Y on stack [Implied]
PLX	Pull X from stack [Implied]
PLY	Pull Y from stack [Implied]
STZ	Store zero [Absolute]
STZ	Store zero [ABS,X]
STZ	Store zero [Zero page]
STZ	Store zero [ZPG,X]
TRB	Test and reset memory bits with accumulator [Absolute]
TRB	Test and reset memory bits with accumulator [Zero page]
TSB	Test and set memory bits with accumulator [Absolute]
TSB	Test and set memory bits with accumulator [Zero page]
BIT	Test Immediate with accumulator [IMMEDIATE]

Instruction Adressing Modes

Op Code	Description
ADC	Add memory to accumulator with carry [(ZPG)]
AND	"AND" memory with accumulator [(ZPG)]
BIT	Test memory bits with accumulator [ABS,X]
BIT	Test memory bits with accumulator [ZPG,X]
CMP	Compare memory and accumulator [(ZPG)]
EOR	"Exclusive Or" memory with accumulator [(ZPG)]
JMP	Jump (New addressing mode) [ABS(IND,X)]
LDA	Load accumulator with memory [(ZPG)]
ORA	"OR" memory with accumulator [(ZPG)]
SBC	Subtract memory from accumulator with borrow [(ZPG)]
STA	Store accumulator in memory [(ZPG)]

Appendix C

Quick Reference Guide to Bugbyter

Bugbyter Command Level

Key	Function
RETURN	Accept user-entered command line.
LEFT ARROW	Move cursor to the left.
RIGHT ARROW	Move cursor to the right.
CONTROL-B	Move cursor to beginning of command line.
CONTROL-C	Accept next keystroke verbatim.
CONTROL-D	Delete a character.
CONTROL-I	Enter insert character mode (any other editor function exits from this mode).
CONTROL-N	Move cursor to end of command line.
CONTROL-X	Delete command line.

General Commands

Command	Function
addressL L	Disassemble code beginning at address (addressL) or continue disassembling (L).
M	Enter Apple Monitor. Return to Bugbyter with CONTROL-Y.
SET	Customize master Bugbyter display where: RIGHT ARROW moves window down. LEFT ARROW moves window up. RETURN fixes subdisplay and advances to next subdisplay.
ON	Turn Bugbyter's Master Display on.
OFF	Turn Bugbyter's Master Display off.
.doscommand	Execute ProDOS command. Press RETURN to return to Bugbyter.
+decimalvalue= -decimalvalue=	Convert positive decimal to hex. Convert negative decimal to hex (65536-decimalvalue).
value= $value=	Convert hex to decimal. Convert hex to decimal.
V	Display copyright and version number.
Q	Quit Bugbyter (exits through ProDOS vector $3DØ).

Register Reference Commands

Command	Function
PC=address	Set 65Ø2 Program Counter with hex address.
A=value	Set 65Ø2 A-register with hex value.
X=value	Set 65Ø2 X-register with hex value.
Y=value	Set 65Ø2 Y-register with hex value.
S=value	Set 65Ø2 Stack pointer with hex value.
P=value	Set 65Ø2 Processor Status register with hex value.
C=value	Set Bugbyter Cycle Counter to value.
R=value	Set Bugbyter keyboard trace rate to value.

Execution Commands

Command	Function
addressG G	Execute code as subroutine at address (addressG) or continue (G). An RTS returns to Bugbyter.
addressJ J	Jump to code at address (addressJ) or continue (J). Used with breakpoints.
addressT T	Enter Trace or Single-Step mode starting at address (addressT) or continue (T). See Trace and Single-Step commands.
addressS S	Enter Trace or Single-Step mode and execute single opcode starting at address (addressS) or continue (S). See Trace and Single-step commands.

Breakpoints

Command	Function
BPn	Set breakpoint n, where: value — sets breakpoint field to value. LEFT ARROW — moves to previous field RIGHT ARROW or SPACE — moves to next field. ESC or RETURN — returns to Bugbyter command line. POINT is a user-defined breakpoint address. COUNT is the number of times the breakpoint address was encountered. TRIG is the user-defined count before breaking. NOTE: To cause a break, TRIG must be set to one or greater. BROKE is the number of times Bugbyter triggered.
IN	Insert BRK (ØØ) op-codes into addresses specified in breakpoint subdisplay. Disables breakpoint modification. (Used for real-time debugging.)
OUT	Replace BRK op-codes with original instructions at addresses specified in the Breakpoint subdisplay. Enables breakpoint modification. (Used for interpretive debugging -- default.)
CLR	Clear all breakpoints.
CLRn	Clear breakpoint n.

Memory Reference

Command	Function
address:	Display 184 memory cells starting at address in hex and ASCII. Use SPACE: to display next 184 cells. Press ESC to return to Bugbyter Master Display.
address:opcode	Assign opcode mnemonic starting at address.
or	
address:value	Fill address with hex value.
or	
address:"text"	Fill address with ASCII character (MSB on).
or	
address:'text'	Fill address with ASCII character (MSB off).
or	
(any mixture)	Multiple values and ASCII text (MSB on or off) can be mixed freely in memory fill. Slash (/) accepts the next character verbatim.
MEM	Edit memory subdisplay where:
	H displays contents of address as hex and ASCII, or
	P displays contents of address & address+1 as pointer.
	address Enter hex address of memory cell(s) to be displayed.
	SPACE or RIGHT ARROW advance to next cell.
	LEFT ARROW return to previous cell.
	ESC return to Bugbyter command line.

Disassembly Options for Trace and Single-Step Modes

Command	Function
O=A	Display 65Ø2 Accumulator in binary.
O=X	Display 65Ø2 X-register in binary.
O=Y	Display 65Ø2 Y-register in binary.
O=S	Display 65Ø2 Stack pointer in binary.
O=P	Display 65Ø2 Processor Status register in binary.
O=B	Display instruction bytes in hex.

O=E	Display computed effective addresses or relative branches and instruction cycles.

Trace and Single-Step Modes

Once in Trace or Single-Step mode (see T or S commands above), Bugbyter responds to the following single keystroke commands:

Command	Function
SPACE	Single step one opcode.
RETURN	Continuous trace.
ESC	Return to Bugbyter command line.
R	Trace until RTS opcode encountered.
RIGHT ARROW	Skip next instruction.
C	Clear Cycle Counter.
P	Use Paddle Ø to adjust trace rate.
K	Use Keyboard Rate (R=value) to adjust trace rate.
Q	Sound off (quiet).
S	Sound on.
1	Display primary Apple screen.
2	Display secondary Apple screen.
T	Display Apple text screen.
L	Display Apple low-resolution graphics screen.
H	Display Apple high-resolution graphics screen.
F	Display full screen graphics.
M	Display mixed text and graphics.

User Soft Switches

Location	Function
start+3	Execute undefined 65Ø2 op-codes ($8Ø=on, ØØ=off {default}).
start+4	Use Paddle Button Ø for trace suspend ($8Ø=on, ØØ=off {default}).
start+5	Use Paddle Ø for trace rate ($8Ø=on {default}, ØØ=off).
start+6	Trace or Single-Step keyboard polling (MSB on + ASCII character code for escape character, MSB off=normal polling {default}).
start+7	Sound ($8Ø=on {default}, ØØ=off).
start+8,+9	Cycle Counter (low, high).
start+$A,+$B	Beginning address of real-time code (default=$FFFF).
start+$C,+$D	Ending address of real-time code (default=$FFFF).

Appendix D

Error Messages

Editor Messages

Two types of errors may appear when you are using the Editor:

- ProDOS errors, indicating that a problem occurred while your source file was read from disk, or while your edited text was being saved to disk.

- Editor command errors, indicating that you typed an unrecognized command or that you typed some command parameter incorrectly.

If a ProDOS error occurs while you are using the Editor, the Editor displays an error message and allows you to correct the problem. If the Editor finds an error in a command that you typed, the Editor displays an appropriate error message and ignores the illegal command. In neither case will you leave the Editor or lose your edited text simply because an error occurred.

ProDOS Errors

Both the Editor and the Assembler use ProDOS to manipulate files that are stored on disk. Thus you may encounter ProDOS errors while using the Editor or Assembler. The Editor displays an error message (similar to the ones found in the BASIC command interpreter) and returns to the Editor Command Level, allowing you to correct the problem and retype the command that caused the ProDOS error.

This Appendix lists the error messages most likely to appear while you are uisng the Editor and Assembler.

BAD PATH/FILE NAME

Indicates that you attempted to create a pathname with illegal characters. ProDOS allows only letters, digits, and periods in pathnames. Each part of a pathname, whether a volume name, directory name, or pathname, must contain no more than fifteen characters. The parts of the pathname must be separated with slashes. No embedded blanks are allowed. For details, see BASIC Programming With ProDOS.

DIRECTORY FULL

Appears if you attempt to save a file to a root directory that is full. Root or volume directories are limited to 55 entries and may not be extended automatically. You can create a subdirectory with the CREATE command and then save your file in that subdirectory.

DIRECTORY NOT FOUND

Indicates that you used a pathname, with the PREFIX command, that is not a mounted disk volume name, or is misspelled.

DISK FULL

Usually appears during a SAVE operation. Repeat the SAVE command, using a disk with more free space.

DISK I/O FAILURE

Typically indicates a bad disk or a disk that is improperly inserted into the drive. If this error occurs during a SAVE operation, your output disk is probably bad and you should do a SAVE command onto another disk. If it occurs during a LOAD operation, your file was only partially read in and may be permanently lost (backing up is wise). If this error occurs for the CATalog, you may have lost your disk directory (a serious problem).

DUPLICATE FILE NAME

Appears if the new pathname you select with the RENAME command already exists as the name of some other file.

FILE LOCKED

Appears if you LOCK your files, then try to SAVE to a LOCKED file. The LOCKed file remains intact, as does your current edit file. UNLOCK the file or change the SAVE name before trying to save the file again. It

is good practice to lock all the files on your disk (except the one you are editing) to avoid losing a file by SAVEing with the wrong name.

FILE NOT FOUND

Indicates that you tried to issue a directory management command (such as LOCK, UNLOCK, DELETE, or RENAME) that cannot find a file, even if the remainder of the pathname and/or the prefix is correct.

FILE SIZE MISMATCH

Indicates that you attempted to SAVE, BSAVE, or XSAVE a sequential file to an existing file of the same type that currently has an end-of-file value larger that the maximum size of the Editor's edit buffer. Since the Editor could not have created such a file, this size conflict suggests that you are attempting to clobber a file that is unrelated to the Editor/Assembler system. You cannot examine the existing file, because it won't fit in the edit buffer.

FILE TOO LARGE

(Can occur for the LOAD, BLOAD, and XLOAD commands.) Indicates that the file you want to load is too large, or that when the file's length is added to the load address the end of the file would extend beyond the edit buffer.

FILE TYPE MISMATCH

Appears if you

- attempt to LOAD some kind of file other than TEXT, or try to SAVE using a name that is already in use by a another type of file, such as Integer or Binary.
- attempt to XLOAD a normal text file or one of the filetypes XLOAD will not attempt to read.
- attempt to BLOAD a file that is not a binary (BIN) file.

PATH NOT FOUND

Indicates that you used a pathname with misspelled or nonexistent subdirectory names in it.

ProDOS FAILURE # => $xx

Indicates a serious failure, either in the Editor/Assembler system or within ProDOS. Before doing anything else, write down as many details as you can about the exact circumstances under which it occurred. You may be able to determine its cause by examining the MLI error message list in the ProDOS Technical Reference Manual . Record the complete configuration of your Apple II system, the type of disks in use, and what commands you used just prior to the problem. This information could be critical to locating a problem in ProDOS or the Editor/ Assembler system. Please report this information to Apple Computer, Inc., preferably in writing, as soon as possible.

THE FILE IS LOCKED. DESTROY ANYWAY (Y/N)?

Appears if you attempt to DELETE a locked file. If you reply by pressing Y, the file will be unlocked and then deleted from the disk directory.

WRITE PROTECTED

Indicates that you attempted to SAVE to a disk that has a write-protect tab.

Editor Command Errors

The Editor checks the syntax, and the validity of the parameters, of each command you type. The command descriptions in Chapter 2 discuss the specific error messages that may appear following a given Editor command.

BAD FORMAT ERROR

You typed the first string field of the Change command as a null string.

BAD RANGE ERROR

The second line number in a range (begin#-end# or begin#-count) has an invalid ending line number. The Copy command requires the second line number of its range parameter to be an existing line in the edit buffer.

BUFFER ERROR

You have attempted to use the ASM command when the edit buffer contains some text lines. The buffer can be SAVEd and then cleared with the NEW command.

CMD SYNTAX ERROR

You typed extra characters following a valid command or following a valid command parameter.

INVALID PARAMETER

A printer or disk slot parameter has referenced an empty slot. (Valid slot numbers are Ø through 7.) Or you tried to select COLumn 59 instead of 4Ø or 8Ø.

MEMORY FULL ERROR

You have filled all available text buffer space. This happens when the Editor attempts to insert a line of text into the buffer; you lose the line of text that you just typed or edited.

MULTI BUFFER ERROR

You have attempted to use the ASM command when both Editor buffers are active. Review the KILL2 command description for details.

NUMERIC OVERFLOW ERROR

A line number was found with a value larger than 63999.

PARAMETER OMITTED ERROR

You neglected to type a necessary parameter for the command you just entered.

SYNTAX ERROR

If this message apears after a Del or Replace command, you typed a range of line numbers with an implied ending line number. If it follows the SAVE command, it indicates that you must type a pathname parameter; no pathname has been previously defined by a LOAD or SAVE command.

UNKNOWN COMMAND ERROR

You have typed a command name that the Editor does not recognize. Usually caused by a misspelling.

Assembler Messages

Two types of errors may occur when you are using the Assembler:

- ProDOS errors indicate that a problem occurred while ProDOS was reading your source program from disk, or while it was writing your object code to disk. When a ProDOS error occurs during an assembly, the Assembler immediately stops the assembly.

- Assembler syntax errors indicate that the Assembler encountered an unrecognized syntax or illegal usage in one of the assembly statements in your source file. The Assembler displays an error message and continues with the assembly.

ProDOS Errors

When a ProDOS error occurs during an assembly, the Assembler stops the assembly, displays the ProDOS error message to indicate the problem, and displays this message on the screen:

```
ASSEMBLY ABORTED.  PRESS RETURN
```

The Assembler then waits for you to press RETURN before it attempts to close any open files. If the Assembler encounters another ProDOS error while trying to close files, it gives up and returns you to the Editor Command Level.

By the Way: Many of the explanations that appear in the Introduction to BASIC Programming With ProDOS are equally applicable to situations in which you are using the Assembler, so you may find it helpful to read that manual if you encounter these errors.

DISK FULL

There is no space left on the output file disk for the output object and/or listing file. This can occur at any time during PASS 2, so be sure there is enough space on the disk for the output object and listing files.

DISK I/O FAILURE

This is usually a read error, but it can also be a write error caused by a bad disk. If it is a hard read error on an input file, the same problem would show up when you try to LOAD that file. If it is a write error on the object file, that file will show in the CATALOG as being only one block, or it will get an I/O error when you try to BLOAD the file into memory.

FILE LOCKED

You have locked an existing copy of the object or listing file that the Assembler is trying to delete so it can create the new one for the current assembly.

FILE NOT FOUND

This error usually occurs because you used the ASM command with a source pathname that does not exist or is not on the disk accessible via the current prefix. It can also occur when a CHN or INCLUDE command is used and the needed file is not present on the proper disk.

WRITE PROTECTED

You have a write-protect tab on the disk to which the Assembler was told to write the output object and/or listing file. This will not occur until the beginning of PASS 2.

By the Way: Many of the error messages relating to pathnames and files (described under Editor Messages) can also occur when you use the Assembler. Keep in mind that the Assembler has one or more input text files and one or two output files active during an assembly, to which any given error message could apply.

If any other ProDOS error messages occur, the Editor/Assembler system has probably been clobbered in memory, because of either a software bug or a hardware failure. Please refer any repeatable errors of this kind to Apple Computer, Inc. in writing; if at all possible send a disk that will reproduce the problem. Include all relevant information: your machine type, memory size, peripheral cards installed, number of disk drives and controllers in use, and any modifications made to any of the above.

You should always attempt to recreate any problem with a fresh copy of the Assembler/Editor system master disk before

concluding that you have found a program bug. Be careful not to use your only copy of any important disks while attempting to recreate a system failure.

Syntax Errors

As the Assembler processes an assembly-language source file, it checks the syntax of each assembly statement and your use of identifiers and operands. The Assembler reads your source statements during each of its two assembly passes. Because different checks are performed during each pass, some error messages will appear only during the first pass; some will appear during the second pass; and some will appear during both passes.

As you watch your assembly, you may want to stop the Assembler if you see an error message that indicates a significant problem in your source file.

At the end of the assembly listing, the Assembler prints an error total. Because this is the sum of all errors encountered during both passes of the assembly, this total may be larger than (even double) the actual number of assembly statements that contain syntax errors needing your correction.

When the Assembler finds an error in a source statement, it displays the appropriate error message and skips to the next assembly statement. For this reason, the Assembler may not indicate all of the syntax errors in your assembly statements the first time you try to assemble your program. The Assembler may still find secondary errors in your source program after you have corrected all of the errors flagged in a previous assembly.

You may sometimes see an Assembler syntax error message that indicates one of a number of possible errors. To identify the specific problem that caused the error, you will have to check your specific source statement.

ADDRESS MODE ERROR

The 65Ø2 does not support all address modes for all opcodes, and this error occurs when an invalid combination of opcode and address mode is found in a statement.

ASSEMBLER PARAMETER ERROR

The Assembler checks the parameters of the ASM command, passed from the Command Interpreter, for correct value ranges. This error message is associated with line Ø of an assembly. There are three possible causes:

- The slot parameters are not within their valid ranges.
- The source filename is longer than thirteen characters and output is generated automatically.
- A pathname more than 32 characters long is used for the output listing file.

BRANCH RANGE ERROR

The 65Ø2 relative branch instruction has a limited addressing range. If the target address of a branch instruction is too far from the branch instruction, this error message is generated in pass two.

BUFFER SIZE ERROR

The SBUFSIZ directive issues this message if the requested new buffer size would reduce the source buffer to a size smaller than the source file currently residing in the buffer. The SBUFSIZ directive should be used to reduce buffer size only from a source file smaller than the desired reduced size.

BYTE OVERFLOW ERROR

The DS directive issues this error if the optional value expression is larger than eight bits. The expression directive characters can be used to ensure that this error is not generated.

DIRECTIVE OPERAND ERROR

This is a catchall error message for many of the assembler directives that require specific operands. Generally, this message indicates that an expression contained some other character where a delimiter was expected. For example, this statement would cause this error:

```
LABEL EQU 33;I FORGET THE SPACE AFTER THE OPERAND
```

The data definition directives check for spaces after commas when an expression list exists in an operand, issuing this error if the spaces are found.

The LST directive's option syntaxer also issues this error if an invalid option character is encountered. The NO prefix for the LST options must be spelled exactly NO or no.

DSECT/DEND ERROR

The DSECT directive begins a dummy section, and the DEND directive terminates it. This error indicates one of three things, depending on the directive in the incorrect statement:

- You attempted to nest the DSECT statement inside an active dummy section (starting a second DSECT within a dummy section).
- You terminated a dummy section with a DEND directive when a dummy section was not active.
- You started a dummy section and did not terminate it with DEND.

DUPLICATE EXT/ENT ERROR

The EXTRN and ENTRY directives check whether a symbol being defined as an external or entry symbol has not already been so defined in a prior EXTRN, ENTRY, or ZDEF statement. Any one symbol can be defined only once as having one of these special characteristics.

In addition to this duplicate directive function, the expression evaluator checks each identifier in an operand expression and issues this error when it encounters a second external identifier in any operand expression.

DUPLICATE IDENTIFIER ERROR

This error occurs when an identifier, in the label field of a statement, is an exact duplicate of a previously defined identifier. Some other name must be chosen for the identifier in the offending statement. It is possible to have multiple duplicates, all or some of which should be unique identifiers. The symbol table dump will not show multiple entries for duplicates flagged with this error.

EQUATE SYNTAX ERROR

The equate directive requires an identifier in the label field; this error occurs if the label field is empty (it doesn't make sense to define nothing as having some value). This error also occurs if the operand expression contains any external identifiers.

EXPRESSION SYNTAX ERROR

The expression evaluator requires that a valid term follow each operator within an expression. A term in an expression is either a constant or an identifier. This error occurs when the text of a statement immediately following an operator, such as the addition operator (+), is not recognized as one of these two items. Terms are recognized by their first character, as follows:

First character	Term type
%	Binary constant
@	Octal constant
$	Hexadecimal constant
"	ASCII character constant
*	Program counter reference
digit	Decimal constant
letter	Identifier (uppercase or lowercase)

This error also occurs if one of the radix characters used to begin numeric constants is not followed by any digits appropriate to that type of constant. For example, %4 causes this error because only Ø and 1 can follow the % (binary) radix character.

EXTRN USED AS ZXTRN WARNING

An identifier, defined as a 16-bit external value by the EXTRN directive, was used as an 8-bit quantity, either as a zero page address or as an immediate operand. This produces nonrelocatable object code if the external value turns out to be an absolute value when a linker attempts to linkage-edit the REL modules together.

The ZXTRN directive should be used in place of EXTRN for this type of identifier, and such identifiers should be defined using the ZDEF directive in the defining module (refer to ZDEF and ZREF for details).

INCLUDE/CHN NESTING ERROR

The INCLUDE directive cannot occur as a statement within an included file.

INDEXING SYNTAX ERROR

The addressing syntax required by the assembler does not allow any character to follow a comma except X or Y for absolute or zero page indexing address mode specification.

INDIRECT REQUIRES ZPAGE ERROR

The Assembler checks the type of an expression in the indirect-indexed and indexed-indirect addressing modes. If the expression is not a zero page expression, this error results (this is a limitation of the 65Ø2 microprocessor).

INDIRECT SYNTAX ERROR

This error has two possible causes:

- Invalid syntax was used for indirect-indexed and indexed-indirect addressing modes. The most common mistake is the improper use of the X and Y registers, for example: LDA (expr,Y) and LDA (expr),X.

- The right parenthesis was omitted in the indirect addressing modes.

INVALID AFTER 1ST IDENTIFIER ERROR

The SBUFSIZ and IBUFSIZ directives allow source and include buffer size management from the assembling program, but they must be used before the first reference or definition of an identifier.

INVALID DELIMITER ERROR

This message indicates incorrect use of delimiters. The expression evaluator requires that all operand expressions terminate with a space or a carriage return character. Various directives require the comma as a delimiter and do not allow spaces after commas. The indexing and indirect addressing syntaxes all require specific character delimiters. The delimiters are the space, the carriage return, the comma, and the parenthesis. When the assembler expects one of these characters and does not find it, this error results.

INVALID FROM INCLUDE/MACRO ERROR

The SBUFSIZ and IBUFSIZ directives may not occur in an include file.

INVALID IDENTIFIER ERROR

This error indicates one of two errors:

- An identifier contains a character not allowed within an identifier (identifiers must begin with a letter and contain only letters, digits, and the period).

- An identifier is not allowed as the operand of a directive, usually because of a previous similar use and some resulting conflicting definition. For example, an identifier cannot be used in the label field of an instruction and be the operand of the EXTRN directive.

OBJ BUFFER CONFLICT ERROR

An OBJ expression was used that is less than the current end of symbol table.

OBJ BUFFER OVERFLOW ERROR

The first OBJ directive in an assembly defines the lower limit of the OBJ buffer for an assembly. If the OBJ buffer fills up to the Assembler code area, this error message results and the assembly is cancelled.

OVERFLOW ERROR

The ASCII source-to-binary conversion routines that convert numeric constants into binary values issue this error message if a constant is entered that generates more than 16 bits of constant. For example, the constant $33333 has too many digits, only four being allowed in a hexadecimal constant.

RELATIVE EXPRSN OPERATOR ERROR

When the REL directive has selected relocatable object-code output, the assembler does not allow a relative address expression or subexpression be divided, multiplied, ANDed, ORed, or exclusive-ORed because doing so causes the result to be nonrelocatable. This prohibition applies only if REL has been used in an assembly.

RESERVED IDENTIFIER ERROR

The Assembler does not allow the labels A,a,X,x,Y,y as identifiers. If you insist on using these identifiers, there is a patch address offset in bytes 3 and 4 of the EDASM.ASM file (using Ø base counting of code bytes) that indicates how far into the object code image to place a patch to cause this checking to be nullified. The patch consists of a CLC ($18) and a RTS ($6Ø) opcode. They should be patched over the first two bytes of a JSR instruction.

SYMBOL/RLD TABLE FULL ERROR

This error occurs when insufficient space is available for the symbol table and, if the REL directive was used, for the relocation dictionary tables. The SBUFSIZ and IBUFSIZ directives default to a total size of 5K, which may be reduced to as little as one page each to provide additional symbol/RLD table space.

UNDEFINED IDENTIFIER ERROR

This error, generated in pass two, indicates that an identifier was referenced but never defined in the assembly source. It is most commonly caused by misspelling an identifier.

UNDEFINED OPCODE ERROR

This error occurs when macros are not enabled (the default condition) and a mnemonic is used that is not found in the assembler's mnemonic table. This is most commonly a spelling error.

>256 EXT/ENT ERROR

When creating the relocation dictionary, the Assembler assigns a unique number (1 to 255) to each EXTRN and ENTRY identifier. The Assembler cancels if you use more than 255 such symbols in your source file.

65Ø2X ADRS MODE/OPCODE ERROR

The Assembler encountered an opcode or an opcode/address-mode combination that is legal only for the 65Ø2X microprocessor. If you are not creating software for a system with the 65Ø2X, you have used an instruction that will not work in your Apple II. Refer to the X65Ø2 directive for more details.

Appendix E

Object File and Symbol Table Formats

Object File Format

The Assembler generates two kinds of ProDOS object files: binary memory-image (BIN) files and RELocatable binary code (REL) files. Unless you use the REL Assembly directive at the start of your assembly-language source file, the Assembler produces a BIN file using the standard ProDOS binary format. The INTERP directive can change the BIN output file type to SYS for creating a ProDOS interpreter file.

You can use the BLOAD command to load Binary files produced by the Assembler, or, if your program is properly coded, you can BRUN your program from the normal BASIC/ProDOS environment (but not from within the Editor Assembler system). To be executable by the BRUN command, your program must begin with executable code, not data, at the lowest address for which object code is generated.

The RELocatable binary file type is recognized by ProDOS but not explicitly supported by any ProDOS commands. Table E-1 defines the format of this file type. The symbol => may be read as "indicates" in this context.

Table E-1. Relocatable File Format

Block	Byte (Hex)	Contents of byte
Ø	Ø	Length of code image, low byte
	1	Length of code image, high byte
Ø to N	2 to c1+1	Binary code image, of length in bytes Ø and 1 above
	2 to c1+2	Begin relocation dictionary (Table E-2), which consists of N 4-byte entries. N is variable. Each four bytes repeat the structure shown in Table E-2.

Table E-2.
Relocatable File Relocation Dictionary Format

Byte	Contents of Byte
1	RLD-flags byte containing 4 flag bits as follows:
	$8Ø bit — Size of relocatable field. SET => 2 byte, Clear => 1 byte
	$4Ø bit — Upper/Lower 8 of a 16 bit value. Set => high 8, Clear => low 8
	$2Ø bit — Normal/reversed 2 byte field. Set => hi-low, Clr => low-hi (the DDB directive causes Set)
	$1Ø bit — Field is EXTRN 16-bit reference. Set => EXTRN, Clr => not EXTRN
	$Ø1 bit — "NOT END OF RLD" flag bit always set ON for RLD entry Clear marks end of RLD
2	Field offset in code, low byte.
3	Field offset in code, high byte
4	Low 8 bits of 16 bit value for an 8-bit field containing upper 8 bits, zero if $4Ø bit clear in RLD byte one. Or, if the $1Ø bit is set, then this is the ESD symbol number.
N*4+1	Binary ØØ marks end of RLD.
N*4+2	Beginning of optional external symbol directory (ESD). This area will only contain bytes if an EXTRN and/or ENTRY directive occurs in the program.
1 to sl	The EXTRN/ENTRY symbolic name of length sl bytes. All bytes have their $8Ø bit set except the last one.
sl+1	Symbol type flag byte defines which type of symbol EXTRN/ENTRY.
$1Ø bit	Set => EXTRN symbol type.

$Ø8 bit	Set => ENTRY symbol type.
sl+2	EXTRN/ENTRY symbol number referred to by an RLD entry with EXT bit set on.
sl+3	High byte of offset for ENTRY type symbol (low is in sl+2).
End mark	Binary zero byte marks end of the ESD entries, of which there may be zero.

Symbol Table Formats

This section describes the symbol table generated during pass one of the assembly, and the table format as it remains after the symbol table has been modified by the symbol-table sort and print routine (pass three). The symbol table will be in its modified form, and the RLD may be clobbered if the symbol table sort and dump is allowed to execute and it overwrites the RLD with its sort index table.

The symbol table is a variable-length entry-format table with flag bits to signal the end of the variable-length-name character string.

When the Symbol Sort and Dump routine executes, it modifies the symbol table format to speed up the scanning of the table for its second phase. The last character of each symbol has its high-order bit set on and the Flagbyte is changed. If the Flagbyte has its $8Ø bit set its value is changed by ORing it with $7E to set all bits on but the $Ø1 bit, which is retained; and the $8Ø bit is set off to mark the end of the Symbolic name. Thus if pass three is run, all Flagbytes will have their $8Ø bits reset, and undefined symbols will have Flagbytes of $7E or $7F.

The basic format is

(Symbolicname)(Flagbyte)(Low value)(High value)

Symbolicname

Consists of 1 to n characters, each with its $8Ø bit set, except that the last character's $8Ø bit is reset.

Flagbyte

Contains the bits that define the characteristics of the symbol and its value and how it can be used to generate instructions, as described below:

$8Ø bit (Undefined-Symbol bit)

This flag means that the symbol was referenced but not defined. This flag is reset when a symbol is defined, and if it remains set at the end of pass one, the symbol is undefined and will cause the NO SUCH LABEL error during pass two. Symbols with this bit set are printed by pass three with an "*" next to the "address" (which is meaningless: it is simply the value of the program counter at the first reference).

$4Ø bit (Unreferenced-Symbol bit)

The symbol was defined but never used as the operand of any instruction in the program. This bit causes a ? to be printed next to the address value for an unreferenced symbol in the dump.

$2Ø bit (Relative-Symbol bit)

The symbol's value is a relative symbol rather than an absolute address. Relative means relative to the beginning of the module. It is used internally by the assembler when generating the Relocatable type of output file to cause an RLD entry to be created for any references to the symbol.

$1Ø bit (EXTeRNal-Symbol bit)

The symbol was defined as an external symbol via the EXTRN directive. This causes the symbol to be put into the ESD and prevents the symbol from being considered undefined, even though no value is assigned to the symbol. Using such a symbol causes an RLD entry to be marked as EXT and causes the external symbol number to be put in the RLD entry in place of the relative offset. EXTeRNal symbols can represent only undefined 16-bit values (not-8 bit or zero page values).

$Ø8 bit (ENTRY-Symbol bit)

The symbol is an entry point into the module that can be referred to by an EXTRN in another module. This causes the symbol to be included in the ESD for resolution by a linkage editor (no linkage editor has yet been implemented).

$Ø4 bit (MACRO-bit)

This symbol is a macro name. The value is the mark position in the transient macro file of the beginning of the macro body's text. This is not yet implemented.

$Ø2 bit (NO-Such-Label Error bit)

The symbol has caused one or more NO SUCH LABEL errors. This is used to prevent a duplication of a single error in the error summary table during pass two.

$Ø1 bit (Forward-Referenced bit)

A forward reference forced the symbol to be considered a 16-bit value. Zero page labels print in the symbol dump with blanks for the first two bytes. They print with two zeros when this bit is set. If the definition is moved forward so that the symbol is defined before it is referred to, reassembling the program will generate shorter, zero page instructions.

Appendix F

Editing BASIC Programs

You can use the Editor to perform many useful editing functions on the text of your BASIC programs. For example, you can use the Find command to locate all statements referring to a given BASIC variable or line number. You can use the global Change command to change all occurrences of a variable name to a new name, or to change all occurrences of a GOSUB to some other line number.

To use the Editor to edit BASIC programs, you must first convert the BASIC program into a sequential text file. BASIC Programming With ProDOS contains a section ("Listing a Basic Program to a File") that shows how to do this. After you have converted the BASIC program into a sequential text file, you can then LOAD the text using the Editor and edit the program as you would any other text file.

When editing BASIC programs, be careful not to change the line number of a statement without also changing all of the references to it elsewhere in the program. It is best to avoid using the Editor to change line numbers within a BASIC program. Instead, use the renumbering function of APA (Applesoft Programmer's Assistant, on the EXAMPLES disk), designed for that purpose. (APA is described in BASIC Programming With ProDOS.)

When you use the Editor to edit a BASIC program, the Editor shows two line numbers on every line. The first is the Editor's relative line number, and the second is the line number of the BASIC statement. Only the line number of the BASIC statement is actually stored in the text file. As you edit, you must use the Editor's relative line numbers as the line number parameters for Editor commands.

When you are finished editing, you must SAVE the text file back onto your disk and reenter BASIC, using the Editor EXIT command. Then, to reenter your edited text into BASIC, type the command

```
EXEC myprogram
```

where myprogram represents the name of your BASIC program text file. This command causes ProDOS to read the entire text file from disk into BASIC, just as if you typed it from the keyboard.

Because BASIC places each line from your text file into a BASIC program according to its line number, each line of your BASIC text must begin with a line number. The order of the lines within your text file is not important; changing the order of the lines without changing their line numbers will not change the way your program will be interpreted by BASIC.

Appendix G

System Memory Use

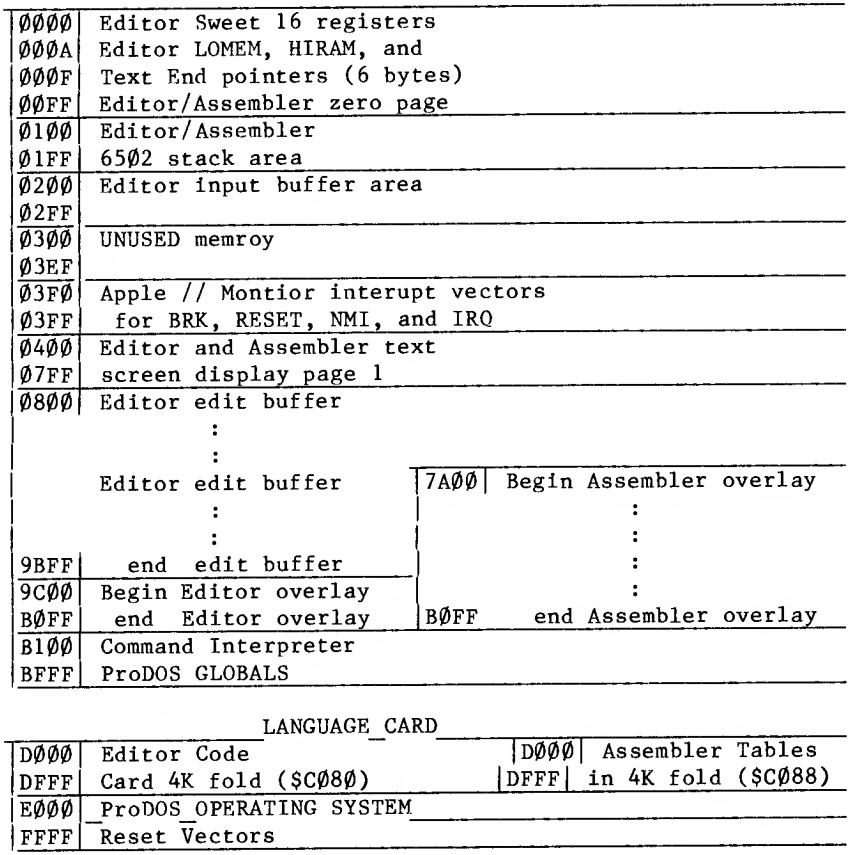
The Editor/Assembler

The memory map in Figure G-1 shows the memory used by the Editor/Assembler. The memory map in Figure G-2 shows where Bugbyter and your own program may reside.

Figure G-1 shows the memory areas used by the various modules that make up the Command Interpreter-Editor and the Assembler. The Assembler shares the same memory space as the all of the Editor and part of the Command Interpreter, or CI. When you invoke the Assembler, the CI overlays the Assembler code over the text buffer and the Editor, as shown in the figure.

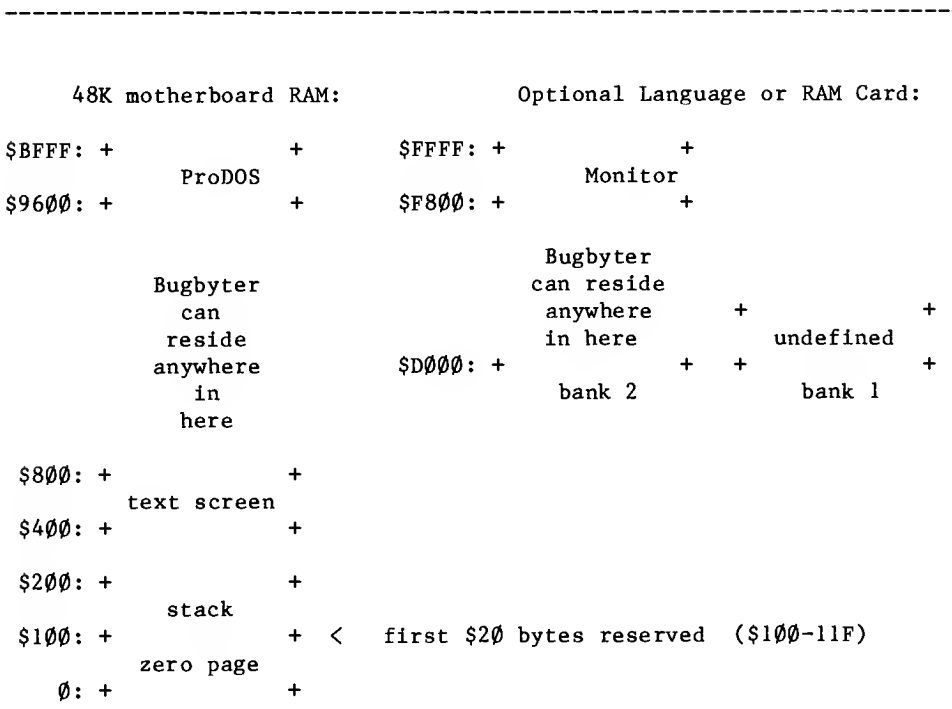
Notice: This memory map of the system is provided for reference only. Apple Computer, Inc. reserves the right to change or expand the areas used at any time and without notice.

Figure G-1. Editor/Assembler Memory Map



The Bugbyter Debugger

Figure G-2. Bugbyter Memory Map



Note: Bugbyter reserves the last 32 bytes of the program stack.

Index

C

E

F

G

H

I

J

K

L

P

Q

R

S

T

U

V

W

X

Y

Z

Apple *II*

ProDOS Assembler Tools

Packing List

This package contains the following items:

Item	Quantity	Part Number	Description
1	1	680-0194	Disk: ProDOS Assembler Tools
2	1	030-0551	Manual: ProDOS Assembler Tools
3	1	030-0136	User Input Report Form
4	1	030-0859	Warranty Card
5	1	825-0623	ProDOS Assembler Tools Spine and Tab Labels
6	1	030-0910	Description Sheet

Peel off label and adhere to spine of binder.

ProDOS
Assembler Tools

ProDOS
Assembler Tools

Peel off labels and adhere to tabs of divider pages.

ProDOS Assembler Tools

ProDOS Assembler Tools

ProDOS Assembler Tools

ProDOS Assembler Tools

In case of questions, contact the dealer from whom you purchased this product.